Automated Software Transplantation

Alexandru Marginean

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

November 8, 2021

I, Alexandru Marginean, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Automated program repair has excited researchers for more than a decade, yet it has yet to find full scale deployment in industry. We report our experience with SAPFIX: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code. We have used SAPFIX at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide. In its first three months of operation, SAPFIX produced 55 repair candidates for 57 crashes reported to SAPFIX, of which 27 have been deem as correct by developers and 14 have been landed into production automatically by SAPFIX. SAPFIX has thus demonstrated the potential of the search-based repair research agenda by deploying, to hundreds of millions of users worldwide, software systems that have been automatically tested and repaired.

Automated software transplantation (autotransplantation) is a form of automated software engineering, where we use search based software engineering to be able to automatically move a functionality of interest from a 'donor' program that implements it into a 'host' program that lacks it. Autotransplantation is a kind of automated program repair where we repair the 'host' program by augmenting it with the missing functionality. Automated software transplantation would open many exciting avenues for software development: suppose we could autotransplant code from one system into another, entirely unrelated, system, potentially written in a different programming language. Being able to do so might greatly enhance the software engineering practice, while reducing the costs.

Automated software transplantation manifests in two different flavors: monolingual, when the languages of the host and donor programs is the same, or multilingual when the languages differ. This thesis introduces a theory of automated software transplantation, and two algorithms implemented in two tools that achieve this: μSCALPEL for monolingual software transplantation and τSCALPEL for multilingual software transplantation. Leveraging lightweight annotation, program analysis identifies an organ (interesting behavior to transplant); testing validates that the organ exhibits the desired behavior during its extraction and after its implantation into a host.

We report encouraging results: in 14 of 17 monolingual transplantation experiments involving 6 donors and 4 hosts, popular real-world systems, we successfully autotransplanted 6 new functionalities; and in 10 out of 10 multlingual transplantation experiments involving 10 donors and 10 hosts, popular real-world systems written in 4 different programming languages, we successfully

autotransplanted 10 new functionalities. That is, we have passed all the test suites that validates the new functionalities behaviour and the fact that the initial program behaviour is preserved. Additionally, we have manually checked the behaviour exercised by the organ. Autotransplantation is also very useful: in just 26 hours computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the VLC media player, a task that is currently done manually by the developers of VLC, since 12 years ago. We autotransplanted call graph generation and indentation for C programs into Kate, (a popular KDE based test editor used as an IDE by a lot of C developers) two features currently missing from Kate, but requested by the users of Kate. Autotransplantation is also efficient: the total runtime across 15 monolingual transplants is 5 hours and a half; the total runtime across 10 multilingual transplants is 33 hours.

Acknowledgements

I would like to express my deep gratitude to my first co-supervisors, Dr. Earl T. Barr and Prof. Mark Harman, for their valuable guidance, help, feedback, and involvement during my PhD research.

I would also like to thank to Dr. Justyna Petke and Dr. Yue Jia, my postdoctoral researcher supervisors, for all their help, guidance, and feedback during my PhD. I am very grateful to the examiners of my thesis: Dr. Emmanuel Letier, the examiner of my first year Viva; Dr. Jens Krinke, my transfer Viva examiner; and Prof. Andreas Zeller and Dr. Federica Sarro, the examiners of my PhD Viva, for their very useful feedback and suggestions for my final thesis. I am thankful to UCL for founding this PhD, through the UCL Studentship.

I would like to thank my wife Kristina for her support. I am grateful to all my colleagues from the CREST center UCL for all the useful discussions, feedback, and support during my PhD. I would like to specially thanks to Prof. William Langdon for all the comments and the guidance he provided during my PhD.

Impact Statement

Our automated repair work can help software engineers by automatically fixing bugs in production code, consisting of millions of lines of code. This allows software engineers to concentrate on more interesting and challenging software engineering task.

A particular case of automated program repair is automated software transplantation that repairs the *host* program that is not implementing a required functionality. Our automated software transplantation work can help software engineers to automatically move a functionality of interest from a donor program into a host program, an activity that is currently largely done manually, is tedious, and error prone. Multilingual software transplantation allows our transplants to cross the language barrier, and thus extend the space of possible donor candidates. This is particularly useful when we want to transplant functionalities from specialized computation area, such as mathematical computations from Fortran code.

The research that this PhD thesis report had impact both in industry and academia. Our *Automated Software Transplantation* [214] initial paper was published in 2015 and has 116 citations[1]. *Automated transplantation of call graph and layout features into Kate* [214] that is an extensive case study of using automated software transplantation on real world systems and was published in 2015 has 23 citations [1]. The paper that introduced our automated repair work, *Sapfix: Automated end-to-end repair at scale* [213], has 61[1] citations and was published in 2019. Our work is the first automated software transplantation approach that treat the entire process of automatically transplanting a functionality of interest from a donor program into a host one. Since published, it opened new areas of research, as the later autotransplantation papers from different research groups show.

The monolingual software transplantation paper that introduced μSCALPEL won the distinguished paper award at ISSTA 2015. In 2016 we used μSCALPEL to automatically transplant H.264 encoder from x264 into VLC, one of the most popular tool for H.264 encoding and media player. Humies is a competition part of the Genetic and Evolutionary Computation (GECCO) conference that provides awards totalling $10,000 for human competitive results that an evolutionary computation approach produced. We participated with the H.264 transplant into Humies and we won the gold medal.

[1]According to Google Scholar on 20 August 2021

The strongest evidence from impact comes from SAPFIX that had a strong industrial impact. SAPFIX is the first ever industrial deployment of automated end-to-end program repair, from test case design through deploying in production. SAPFIX is the largest scale deployment of any search based software engineering since Sapienz that was used at Facebook to automatically repair 6 production systems, each consisting in tens of millions of lines of code and which are collectively impacting directly billions of people. The Facebook engineering blog post[2] about Sapfix says:

> *"Debugging code is drudgery. But SapFix, a new AI hybrid tool created by Facebook engineers, can significantly reduce the amount of time engineers spend on debugging, while also speeding up the process of rolling out new software. SapFix can automatically generate fixes for specific bugs, and then propose them to engineers for approval and deployment to production."*

[2]`https://engineering.fb.com/2018/09/13/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the main topics of this PhD thesis: automated program repair (Chapter 3); and a particular kind of program repair, the main topic of this thesis: automated software transplantation (Chapter 4 and Chapter 5) that is automatically moving a functionality of interest from a program that implements it (*Donor*) into a program that lacks it (*Host*). Automatic software transplantation repairs the functional characteristics of the host program, by augmenting its functionality with a required functionality that we call *Organ* and that was initially missing.

We report an automated program repair framework in SAPFIX that is capable of automatically producing and landing into production fixes for systems on millions of lines of code that are used by billions of people around the word. We also report tools, algorithms, and theory for solving the problem of automated software transplantation across the same programming language (*monolingual*) and across different programming languages (*multilingual*). μSCALPEL, our monolingual software transplantation tool and all our monolingual software transplantation experiments are available open source at `http://crest.cs.ucl.ac.uk/autotransplantation`.

This chapter is organized as follows: Section 1.1 introduces our automated program repair approach; Section 1.2 introduce the automated software transplantation problem, a particular kind of automated program repair; Section 1.3 presents a motivating example for our automated software transplantation work; Section 1.4 reports the main contributions of this PhD thesis; Section 1.5 reports the publications during this PhD, the most them being included in chapters in this thesis; and finally, Section 1.6 shows the organization of this PhD thesis.

1.1 Automated Program Repair

Automated program repair traditionally seeks to find small changes to software systems that patch known bugs [184, 247]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search-based program repair [185]. In scientific evaluations of previous approaches to automated patching, this test suite is usually taken as a given, it is often constructed by developers [228].

Recently, the automated test case design system, Sapienz [210], has been deployed at scale [6, 132]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have been able to find fixes in approximately 75% of cases [6], indicating a high signal-to-noise ratio [132] for Sapienz bug reports. Nevertheless, developers' time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

The deployment of Sapienz automated test design means that automated repair can now also take advantage of automated software test design to automatically re-test candidate patches. In this thesis we tackle the problem of deploying simple automated program repair strategies that are scalable and efficient from the point of view of execution time and resource usage. We deployed such automated repairs in the continuous integration system of Facebook. Our approach automatically produces candidate fixes, validates them with the automatically generated test cases that Sapienz designed, and automatically lands them in production, once the final gatekeeper that is a human engineer approves the fixes.

This is the first time when automated program repair has been deployed at such scale in industry. Our approach runs across 6 major Facebook Android applications: Facebook, Messenger, Instagram, Workchat, Workplace, and FBLite. These systems are are some of the largest, most complex, and most widely used applications in Google Play. They consists in millions of lines of codes and are used by billions of people across the word. Our tool for automated program repair, SAPFIX, successfully automatically produced, validated, and landed in production fixes for null pointer exception crashes across all these applications. Thus, all the users of these applications run today on their mobile phone code that was automatically generated by our automated program repair approach.

1.2 Automated Software Transplantation

Our work in SAPFIX (Chapter 3) tackles the problem of automatic program repair in case of known bugs that manifest as crashes. Such patches are usually small and simple as we do not seek to change the good functionality of the program, but just to make the crash not manifest any more. A different kind of automated program repair that is the main topic of this PhD thesis is automated program repair for functional characteristics of a software system: *automated software transplantation* is a particular kind of automated program repair, where we aim to repair the functionality of a program, by augmenting it with some functionality of interest that the program should implement but is not currently doing so.

Curently software engineers and developers spend a great deal of time extracting, porting, and rewriting existing code to extend the functionality of existing software systems. Currently this is a tedious, laborious, slow and error prone manual activity [63]. The recent advances and developments in the world of app stores increase the need of software engineers to port functionality across different applications, potentially across different platforms and programming languages. Under a manual examination of the functionality offered by the 50 most popular apps in the iOS App Store we can see that 42 out of these also exist in Google Play app store. In each case, app developers are reimplementing near-identical functionality for each platform. The cost of developing each such app has been estimated to reside in the range \$0.5–\$1 million [348], suggesting that duplication is a highly non–trivial cost; one that would be better avoided where possible.

Automated software transplantation is the problem of automatically transferring code from one program that we call the donor into another one that we call the host. The host and the donor systems are potentially entirely unrelated systems, and possibly written in different programming languages. Automated software transplantation repairs the functionality of the host system by augmenting it with the functionality that the transferred code from the donor implements and that was previously missing in the host system. Such an automated software transplantation approach would help at removing this burden from the software engineers, as well as reducing the general, potentially huge, cost of implementing already existing functionalities. The benefits are potentially huge as nowadays we have access to a lot of already implemented functionalities on open source code repositories such as GitHub[1], sourceforge[2], or Bitbucket[3]: Markovtsev *et al.* [215] identified an exponential increase in the number of open source repositories of GitHub. The public Git Archive they released contains 182,014 top-bookmarked Git repositories from GitHub and occupies three TB disk space, a huge source of existing functionalities.

The software engineering literature provides analyses and partial support for manual code reuse, for example: clone detection [35, 151, 170, 327], code migration [176, 318], code salvaging [60], reuse [61, 62, 68], dependence analysis [32, 119, 133, 140] and feature extraction techniques [123, 163, 190]. However, the overall process of transplanting a useful, non-trivial functionality from a *donor* system, into a target system of transplantation, which we call *host*, is still largely unautomated.

In this thesis we want to answer the following question: Can we automate the process of extracting functionality from the *donor* system and transplant it into a potentially totally unrelated *host* system? What if the languages of the host and of the donor differ? This PhD thesis introduces theories, algorithms, and tools for automated software transplantation. Our current approach uses lightweight manual annotations for identifying an organ's entry (interesting behavior to transplant), and an implantation point into the host.

We use static analysis to identify an over-approximation of the organ, to extract it, and to construct its dependencies. We use a novel form of in-situ unit testing for capturing the desired behaviour of the organ. We use genetic programming (GP) for transplanting, translating, and adapting the organ to the host, under the in-situ unit testing. Regression and acceptance tests are used for validating that the organ exhibits the desired behaviour when running in the host. This thesis tackles both the problem of monolingual software transplantation (*i.e.* software transplantation when both the

[1] `https://github.com/`
[2] `http://sourceforge.net/`
[3] `https://bitbucket.org/`

host and the donor are written in the same language) and multilingual software transplantation (*i.e.* software transplantation when the languages of the host and of the donor differ). The idea of using search based techniques for software transplantation, as we do in this thesis, was first proposed (but unimplemented and unevaluated) by Harman *et al.* in the keynote of the 2013 Working Conference on Reverse Engineering [129].

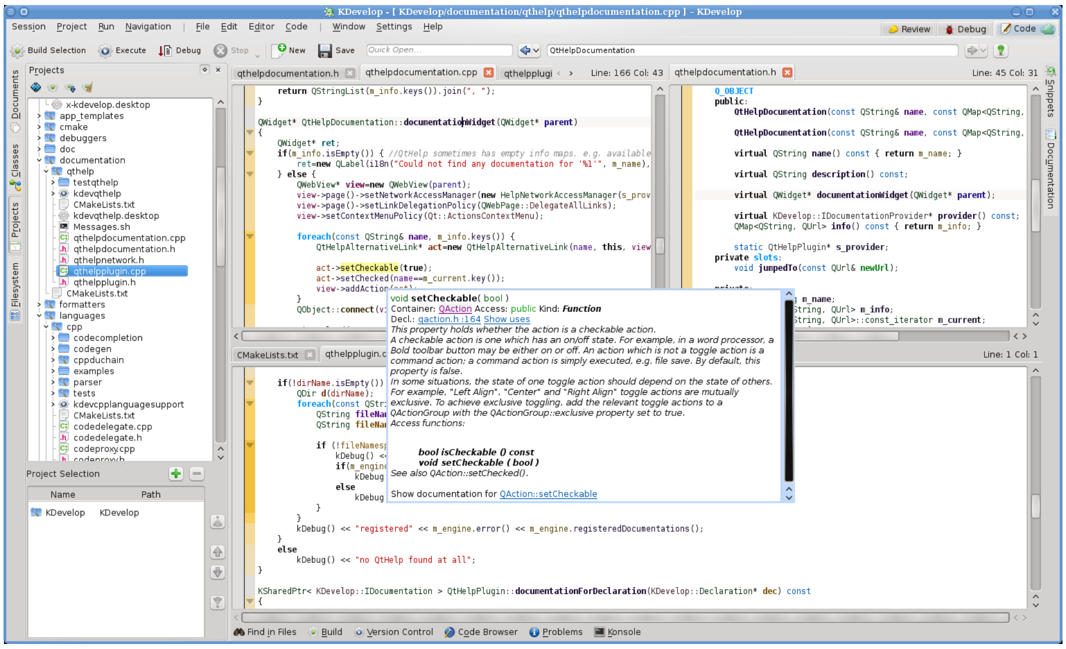
An user of our autotransplantation tools must first identify the entry point of the organ (interesting behaviour of the *donor* that the user wants to transplant). Then, the user must provide the implantation point into the *host*, which is the point where the organ is added, and from where the organ takes its inputs from the *host* system. Having this manual annotations, our goal is to identify, and extract an *organ* (all code associated with the feature of interest), and make it compatible with the implantation point in the *host*. For this the organ might be transformed, as it is pointed out is Section 4.4. We also require test suites that guide the GP algorithm in its search for donor code modifications required to make it fully executable, and to exhibit the desired behaviour (according to the test suites) when implanted into the *host* system.

The problem we are tackling is hard: alien code transplanted from a *donor* into a totally unrelated, potentially written in a different programming language *host* is unlikely to even compile in the host's environment. More than this initial requirement, we impose for our organ to pass three validation test suites, and one test suite used to guide the GP process. The extraction of the organ involves identifying and extracting all semantically required code for the organ to execute: initializations, data structures, operators, functions, preprocessor constructions, etc. After extraction, this alien code is inserted into the host system. For this, there are required nontrivial modifications to the organ to ensure that it adds the required feature, that is does not break the existing functionality, and that it is compatible with the implantation point from the host (namespace conflicts may arise for example). Our initial experiments with autotransplantation show the feasibility of our approach, and its efficiency. Autotransplantation also shows to be useful, by transplanting organs that were manually maintained by the developers of the hosts, or required by the user of the hosts.

Feature identification is well studied [62, 68, 318]. Extracting a component of a system, given the identification of a suitable feature is also well studied, through work on slicing and dependence analysis [119, 123, 133, 163, 190]. The challenges of automatically extracting a feature, from one system, transplanting it into an entirely different system, and usefully executing it in the organ beneficiary, however, have not previously been studied in the literature.

1.3 Motivating Example

This section motivates the software transplantation problem with two example transplants between big, non-trivial, real world programs. The functionalities that we transplanted were requested by the users of the host system: the system into which we transplant. Section 4.11 details these two transplants.


KDevelop - [KDevelop/documentation/qthelp/qthelpdocumentation.cpp] - KDevelop

```
Projects
  x-kdevelop.desktop
  app_templates
  cmake
  debuggers
  doc
  documentation
    qthelp
      testqthelp
      kdevqthelp
      CMakeLists.txt
      kdevqthelp.desktop
      Messages.sh
      qthelpdocumentation.cpp
      qthelpdocumentation.h
      qthelpnetwork.h
      qthelpplugin.cpp
      qthelpplugin.h
    CMakeLists.txt
  formatters
  languages
    cpp
      codecompletion
      codegen
      cppduchain
      examples
      parser
      tests
      kdevcpplanguagesupport
      CMakeLists.txt
      codedelegate.cpp
      codedelegate.h
      codeproxy.cpp
      codeproxy.h

Project Selection
Name        Path
KDevelop    KDevelop
```

qthelpdocumentation.cpp — Line: 166 Col: 43

```
    return QStringList(m_info.keys()).join(", ");
}

QWidget* QtHelpDocumentation::documentationWidget(QWidget* parent)
{
    QWidget* ret;
    if(m_info.isEmpty()) { //QtHelp sometimes has empty info maps. e.g. available
        ret=new QLabel(i18n("Could not find any documentation for '%1'", m_name),
    } else {
        QWebView* view=new QWebView(parent);
        view->page()->setNetworkAccessManager(new HelpNetworkAccessManager(s_prov
        view->page()->setLinkDelegationPolicy(QWebPage::DelegateAllLinks);
        view->setContextMenuPolicy(Qt::ActionsContextMenu);

        foreach(const QString& name, m_info.keys()) {
            QtHelpAlternativeLink* act=new QtHelpAlternativeLink(name, this, view

            act->setCheckable(true);
            act->setChecked(name==m_current.key());
            view->addAction(act);
        }
        QObject::connect(vi
```

qthelpdocumentation.h — Line: 45 Col: 31

```
    Q_OBJECT
    public:
        QtHelpDocumentation(const QString& name, const QMap<QString,

        QtHelpDocumentation(const QString& name, const QMap<QString,

        virtual QString name() const { return m_name; }

        virtual QString description() const;

        virtual QWidget* documentationWidget(QWidget* parent);

        virtual KDevelop::IDocumentationProvider* provider() const;
        QMap<QString, QUrl> info() const { return m_info; }

        static QtHelpPlugin* s_provider;
    private slots:
        void jumpedTo(const QUrl& newUrl);

    private:
        m_name;
        String, QUrl> m_info;
        String, QUrl>::const_iterator m_current;
```

void **setCheckable**(bool)
Container: QAction Access: public Kind: ***Function***
Decl.: qaction.h :164 Show uses
This property holds whether the action is a checkable action.
A checkable action is one which has an on/off state. For example, in a word processor, a Bold toolbar button may be either on or off. An action which is not a toggle action is a command action; a command action is simply executed, e.g. file save. By default, this property is false.
In some situations, the state of one toggle action should depend on the state of others. For example, "Left Align", "Center" and "Right Align" toggle actions are mutually exclusive. To achieve exclusive toggling, add the relevant toggle actions to a QActionGroup with the QActionGroup::exclusive property set to true.
Access functions:

bool isCheckable () const
void setCheckable (bool)
See also QAction::setChecked().

Show documentation for QAction::setCheckable

qthelpplugin.cpp — Line: 1 Col: 1

```
    if(!dirName.isEmpty())
        QDir d(dirName);
        foreach(const QStr
            QString fileNa
            QString fileNa

            if (!fileNamesp
                kDebug() <<
                if(m_engine
                    kDebug
                else
                    kDebug
            }
        }
        kDebug() << "registered" << m_engine.error() << m_engine.registeredDocumentations();
    }
    else
        kDebug() << "no QtHelp found at all";
}

KSharedPtr< KDevelop::IDocumentation > QtHelpPlugin::documentationForDeclaration(KDevelop::Declaration* dec) const
{
```

Session Project Run Navigation | File Edit Editor Code | Window Settings Help
Review Debug Code
Build Selection Execute Debug Stop New Save Quick Open... QtHelpDocumentation
Documents Classes Projects Filesystem Snippets Documentation
CMakeLists.txt qthelpplugin.c
Find in Files Build Version Control Code Browser Problems Konsole

Figure 1.1: Host Kate system. Initially, Kate does not implement neither C call graph generation or C code indentation functionalities.

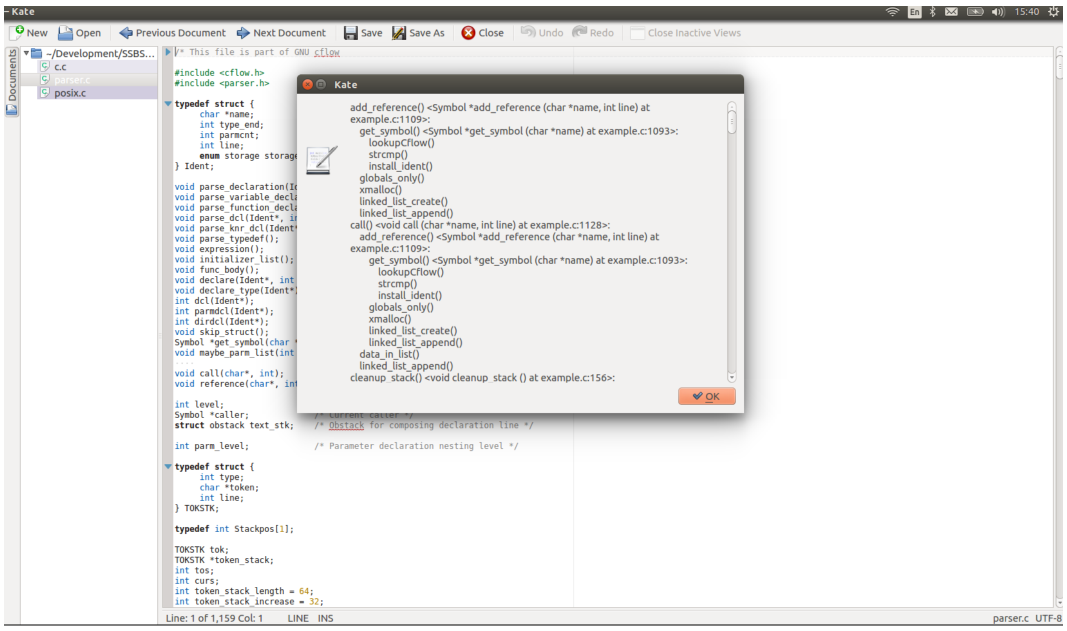
Kate[4] is a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. The user community of Kate observed that Kate does not implement two useful features for a C IDE and requested them on Kate's development forum: C call graph generation and C code indentation. Figure 1.1 shows the GUI of the original Kate system that does not produce call graphs for C programs and does not indent C code. We call this the host Kate.

We used automated software transplantation to implement these two features currently missing from Kate: call graph generation and automatic layout for C programs, which have been requested by users on the Kate development forum. Our approach uses a lightweight annotation system with Search Based techniques augmented by static analysis for automated transplantation. The results are promising: on average, our tool requires 101 minutes of standard desktop machine time to transplant the call graph feature, and 31 minutes to transplant the indentation feature.

Guided by dependence analysis and testing, our approach uses a variant of genetic programming to identify and extract useful functionality from a donor program, and transplant it into a (possibly unrelated) host program. This example illustrates the way in which realistic, scalable, and useful real-world transplantation can be achieved using automated software transplantation.

In the first transplant example, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs. This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, the developer merely needs to identify the entry point of

[4]`http://kate-editor.org`



```
/* This file is part of GNU cflow

#include <cflow.h>
#include <parser.h>

typedef struct {
     char *name;
     int type_end;
     int parmcnt;
     int line;
     enum storage storage
} Ident;

void parse_declaration(Id
void parse_variable_decla
void parse_function_decla
void parse_dcl(Ident*, in
void parse_knr_dcl(Ident*
void parse_typedef();
void expression();
void initializer_list();
void func_body();
void declare(Ident*, int
void declare_type(Ident*)
int dcl(Ident*);
int parmdcl(Ident*);
int dirdcl(Ident*);
void skip_struct();
Symbol *get_symbol(char *
void maybe_parm_list(int

void call(char*, int);
void reference(char*, int

int level;
Symbol *caller;             /* Current caller */
struct obstack text_stk;    /* Obstack for composing declaration line */

int parm_level;             /* Parameter declaration nesting level */

typedef struct {
     int type;
     char *token;
     int line;
} TOKSTK;

typedef int Stackpos[1];

TOKSTK tok;
TOKSTK *token_stack;
int tos;
int curs;
int token_stack_length = 64;
int token_stack_increase = 32;
```

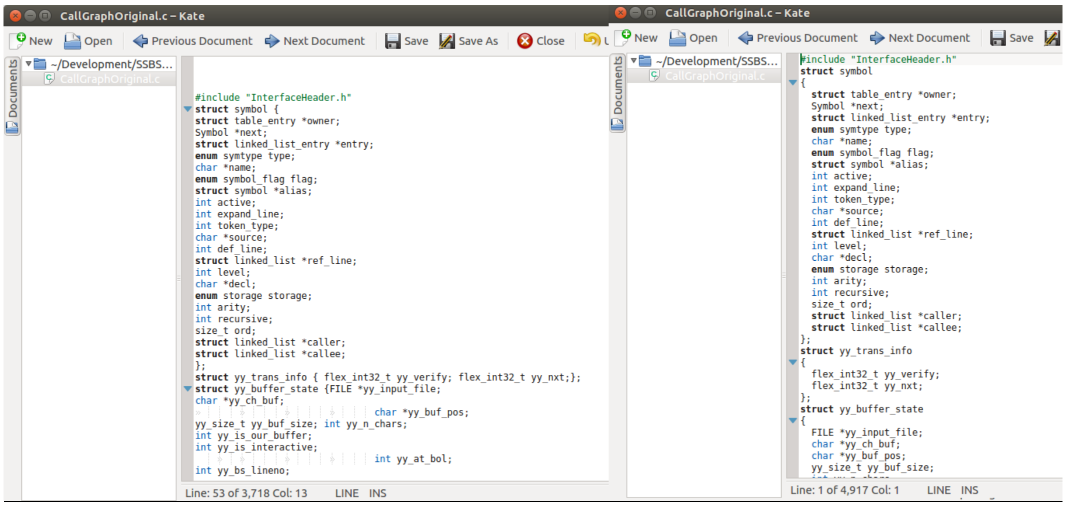
```
add_reference() <Symbol *add_reference (char *name, int line) at
example.c:1109>:
  get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
    lookupCflow()
    strcmp()
    install_ident()
  globals_only()
  xmalloc()
  linked_list_create()
  linked_list_append()
call() <void call (char *name, int line) at example.c:1128>:
  add_reference() <Symbol *add_reference (char *name, int line) at
example.c:1109>:
    get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
      lookupCflow()
      strcmp()
      install_ident()
    globals_only()
    xmalloc()
    linked_list_create()
    linked_list_append()
  data_in_list()
  linked_list_append()
cleanup_stack() <void cleanup_stack () at example.c:156>:
```

Figure 1.2: Beneficiary Kate system that after transplantation is able to produce call graphs for C programs. The front window shows the call graph for the C source code opened in the back window. Our transplant simply displays the call graph in a new window in Kate. For a deeper integration, capable of jumping to source, we would either need human written code, or a new transplant from a donor of jump to definition functionality.

the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [253], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in this thesis, but believe that existing achievements in Search Based [125] and other [55] test data generation techniques will help us to ensure that this reliance is reasonable and practical. Figure 1.2 shows the Kate *beneficiary* system after the transplant of the C call graph generation functionality: Kate can now generate the call graph for the active C source file.

Although we autotransplanted the call graph generation functionality from cflow into Kate, one could imagine deeper integrations of the call graph functionality in Kate. Currently, our transplant will display the call graph in a new window when actioned (Figure 1.2). Ideally clicking on a call in the call graph would open the source code of that function in Kate. Unfortunately such functionality (*i.e.* opening the source code of the function when clicking on it) does not exist in the cflow code. Section 4.5.3 describes this problem in details and provide some possible solutions. Currently, to achieve such deeper integration, a human could write the code to jump to definition or alternatively, the user of autotransplantation could identify another donor program that provides such functionality and execute a new transplant after the one from cflow.

Our second transplantation incorporates a pretty printer for C, which `Kate` only partially supports and which its users have requested. Figure 1.3 shows the *beneficiary* Kate system after the transplant



```
#include "InterfaceHeader.h"
struct symbol {
struct table_entry *owner;
Symbol *next;
struct linked_list_entry *entry;
enum symtype type;
char *name;
enum symbol_flag flag;
struct symbol *alias;
int active;
int expand_line;
int token_type;
char *source;
int def_line;
struct linked_list *ref_line;
int level;
char *decl;
enum storage storage;
int arity;
int recursive;
size_t ord;
struct linked_list *caller;
struct linked_list *callee;
};
struct yy_trans_info { flex_int32_t yy_verify; flex_int32_t yy_nxt;};
struct yy_buffer_state {FILE *yy_input_file;
char *yy_ch_buf;
                                char *yy_buf_pos;
yy_size_t yy_buf_size; int yy_n_chars;
int yy_is_our_buffer;
int yy_is_interactive;
                                int yy_at_bol;
int yy_bs_lineno;
```

Line: 53 of 3,718 Col: 13 LINE INS

```
#include "InterfaceHeader.h"
struct symbol
{
  struct table_entry *owner;
  Symbol *next;
  struct linked_list_entry *entry;
  enum symtype type;
  char *name;
  enum symbol_flag flag;
  struct symbol *alias;
  int active;
  int expand_line;
  int token_type;
  char *source;
  int def_line;
  struct linked_list *ref_line;
  int level;
  char *decl;
  enum storage storage;
  int arity;
  int recursive;
  size_t ord;
  struct linked_list *caller;
  struct linked_list *callee;
};
struct yy_trans_info
{
  flex_int32_t yy_verify;
  flex_int32_t yy_nxt;
};
struct yy_buffer_state
{
  FILE *yy_input_file;
  char *yy_ch_buf;
  char *yy_buf_pos;
  yy_size_t yy_buf_size;
```

Line: 1 of 4,917 Col: 1 LINE INS

Figure 1.3: Beneficiary Kate system that after transplantation is able to indent C source files. The left screen shows the unindented source code. The right screen shows the indented show code that our transplant produced. Our transplant indents the code in the currently opened window.

of the C indentation functionality: Kate can now indent C source code. At the time of writing, we deployed a new version of `Kate` that incorporates these features. We hope to be able to report on the uptake of this 'genetically improved' `Kate`. Section 4.11 explains in detail this experiment and presents our results. The current indentation functionality that we transplanted will indent the code once the plugin is actioned. One could imagine a slightly different functionality, such as automatically indenting when writing in Kate. Our autotransplantation approach could do such transplant given different inputs provided to our autotransplantation tool as we describe in Section 4.11.

An alternative to both these transplants would be to directly call the donors (GNU indent and GNU cflow) from Kate, through the CLI. Since in these cases, the functionalities that we seek to transplant are the vast majority of the donor's source code, an user of Kate could do so. In general, the functionalities that we seek to transplant are just a small part of the donor system and the donor systems are not always CLIs. This makes the usage of the organs directly from the donor not feasible in general. Section 4.5.4 discuss in more details using the organs directly from the donor.

Even in the case of our two Kate transplants, using them from CLI would not be so convenient for an user of Kate. In the case of our indent transplant, our transplant indents the currently opened source file in Kate. To use indentation directly from our donor, an user would have to manually call it from CLI, provide the path for the file to indent, and finally copy and paste the output of GNU indent into the Kate's window. Our transplant does all of this with a single click in Kate's interface Similarly, in the case of the cflow transplant, to directly call GNU cflow, the user would have to manually type in the CLI the cflow command together with its arguments. Our cflow transplants does all of this with a single click in Kate's user interface.

Our work is closely related to recent achievements in genetic improvement, which have been able to dramatically speed up real world systems [178, 289, 293], port between languages [176], balance memory consumption and execution time [342], reduced energy consumption [51, 280] and fix bugs [184]. Most closely related to our approach is work on auto-specialisation using transplantation [248] and grow and graft genetic improvement [127]. Whereas auto-specialisation transplants from different versions of the same system (or closely related systems) and grow and graft transplants newly grown simple features, autotransplantation transplants large-scale features (and subsystems) from one or more donors into an unrelated host.

1.4 Current Contributions

The main contributions of the current PhD research are:

1. The first industrial implementation and deployment of automated program repair on systems of millions of lines of code, used by over two billions people in SAPFIX (Chapter 3). SAPFIX is currently deployed and automatically fixes bugs over all Facebook's Android applications: Facebook, Instagram, Messenger, FBLite, WorkChat, and WorkPlace. The automatically generated fixes are exercised daily by millions of people that use Facebook's Android applications.

2. The formalization of the automated software transplantation problem (Section 4.3);

3. A comprehensive literature survey of the software transplantation literature and the relevant areas in the software engineering literature (Chapter 2);

4. The first monolingual software transplantation algorithm μTrans and its realization in μSCALPEL (Chapter 4);

5. The first multilingual software transplantation algorithm τTrans and its realization in τSCALPEL (Chapter 5);

6. A demonstration of μSCALPEL's and τSCALPEL's feasibility, effectiveness, and usefulness over non-trivial real-world programs, at transplanting software organs — useful, nontrivial functionality — from donors to hosts (Section 4.8, Section 4.9, Section 4.11, Section 5.5, Section 5.6);

1.5 Publications

Content from the following publications appears in this PhD thesis:

1. **Automated Software Transplantation** [30] was published in ISSTA 2015 and currently has 116 citations according to Google Scholar on 20 August 2021 (Chapter 4);

2. **Automated Software Transplantation Artifact Evaluation** was published in ISSTA 2015 Artifact Evaluation Track[5] (Chapter 4);

[5]http://issta2015.cs.uoregon.edu

3. **Automated transplantation of call graph and layout features into Kate** [214] was published in SSBSE 2015 and currently has 23 citations according to Google Scholar on 20 August 2021 (Section 4.11);

4. **Automated software transplantation of H.264 encoder** is our Humies 2016 entry that won the gold medal[6] (Section 4.10);

5. **Sapfix: Automated End-to-End Repair at Scale** [213] was published in ICSE SEIP 2019 and currently has 61 citations according to Google Scholar on 20 August 2021 (Chapter 3));

During my PhD I undertook additional research that is not part of this thesis:

1. **Grow and serve: Growing Django citation services using SBSE** was published in SSBSE 2016 and currently has 18 citations according to Google Scholar on 20 August 2021;

2. **Indexing Operators to Extend the Reach of Symbolic Execution** was published on arxiv.org[7] and currently has 1 citation according to Google Scholar on 20 August 2021 ;

1.6 Roadmap

This thesis is organized as following: Chapter 2 discuss the relevant background knowledge for our automated program repair and automated software transplantation work; Chapter 3 presents SAPFIX, our tool for automated program repair at Facebook's scale; Chapter 4 presents our monolingual automated software transplantation approach; Chapter 5 presents our multilingual automated software transplantation approach; Chapter 6 discuss future works areas for the research that this thesis report; while Chapter 7 concludes this thesis.

Finally, the appendixes report other work being undertaken during this PhD, but outside the scope of this thesis: *Grow and Serve*; *Indexing Operators to Extend the Reach of Symbolic Execution*; and *Retype: Changing Types to Change Behaviour*.

[6] http://gecco-2016.sigevo.org/index.html/Humies.html
[7] https://arxiv.org/abs/1806.10235

Chapter 2

Background

This chapter introduces some background knowledge relevant for our automated program repair work, with a focus on automated software transplantation and surveys the relevant software engineering literature.

We put our work into the wider research context, discussing how it arises out of Search Based Software Engineering (Section 2.2) and *Genetic Improvement* (Section 2.3) in particular; and how it leverages work on evolutionary computation (Section 2.1), program slicing (Section 2.4), code synthesis (Section 2.5) program translation (Section 2.6), and code search (Section 2.7). Next, we discuss relevant transplant maintainability approaches (Section 2.8) and relevant work in automated fault finding (Section 2.9).

Section 2.10 discusses the grow and serve approach that is another technique where the contributions of the thesis proved to be useful. Section 2.11 introduces the state of the art tools on which this research builds on. Section 2.12 reviews the related automated program repair literature. Next, we examine, more closely, the existent related automated software transplantation approaches in Section 2.13. Finally, Section 2.14 concludes this chapter.

2.1 Evolutionary Computation

Automated software transplantation augments the functionality of the *host* program and thus, improves it. Automated software transplantation is a form of genetic improvement (GI) that improves the *functional* properties of the *host* program. Our GI approach uses a form of evolutionary computation in the transplantation process: genetic programming. This section introduces the area of evolutionary computation and details the genetic programming approach, an evolutionary computation technique.

Evolutionary computation [24] is an area in software engineering that is inspired from the natural evolutionary process. In his seminal work in 1859, Charles Darwin defines the biological evolution [73] as the nature's process through which it assures the evolution of the biological organisms across multiple generations. Darwin's work is the foundation of the evolutionary biology [47] that states that populations of biological organisms evolve over the course of multiple generations through the course of natural selection. The natural selection assures that in nature, the biological organisms that adapt the best to the environment are the ones to survive. The evolutionary computation approach applies the same ideas in software engineering: it evolves populations of solutions (or *individuals*) across multiple generations in the hope that in the final generation we will have the best solutions.

The process of the biological evolution [95] *mutates* the genes of the individuals and / or combines the genes of two individuals by the means of sexual recombination or *crossover*. The natural selection process allows only the *fittest* individuals to survive to the next generation. The evolutionary algorithms mimic this process. Alan Turing was the first to propose an evolutionary approach to computer science problems [3] as early as 1948, but he did not expand on it. The evolutionary computation idea remained unexplored until 1975 when John Holland published his seminal work “Adaptation in natural and artificial systems: an introductory analysis with application to biology” [139].

Biological evolution [73] is a method to search in an enormous search space of possible gene sequences for the best combinations that allow the organism to survive and adapt to its environment. Similarly, evolutionary computation searches in an enormous search space for the optimal solutions. In general, an evolutionary algorithm does not produce the best solution because of the resource limits (*i.e.* an evolutionary algorithm stops after a predefined number of generations). The solution that the evolutionary algorithms finds is “good enough” [292].

The problems amendable to evolutionary computation usually have a huge search space of possible solution, out of which we have to search for *good enough* solutions. These problems cannot be easily solved in polynomial time (*e.g.* NP-Complete problems [1]) and exhaustive search is unfeasible in practice because of the exponential run-time [340] required to explore the entire search space of solutions. For example, consider the traveling salesman problem (TSP) [137]:

“*Given a list of n cities and a cost matrix where $c_{i,j}$ represents the cost of traveling from city i to city j, what is the most cost effective route that a traveling salesman can take to visit exactly ones each of the cities? The traveling salesman can chose its starting point.*”

TSP [66] is a NP-hard problem [153] and thus, no efficient solution does currently exist that finds the optimal route. To find the most effective route, the only *known* way is the exhaustive search that is to evaluate all the potential routes. But how expensive is it to evaluate all of them? Let n be the number of cities that the traveler salesman wants to visit. First, he or she has to try each of the n cities to identify the *best* starting point. For each of them, the traveler salesman has $n-1$ choices for the second city to visit: $n*(n-1)$ routes for the first two cities. Going further, to visit all the n cities there are $1*2*\cdots n$ routes in total that is a complexity of $O(n!)$. This complexity is unfeasible in practice when n becomes greater than 10. Evolutionary computation can help here: although it does not guarantee to obtain the best solution, evolutionary computation leads to acceptable solutions in a reasonable time. EC techniques such as ant colony optimization [90], genetic algorithms [182], or cuckoo search [238] were applied to tackle the TSP.

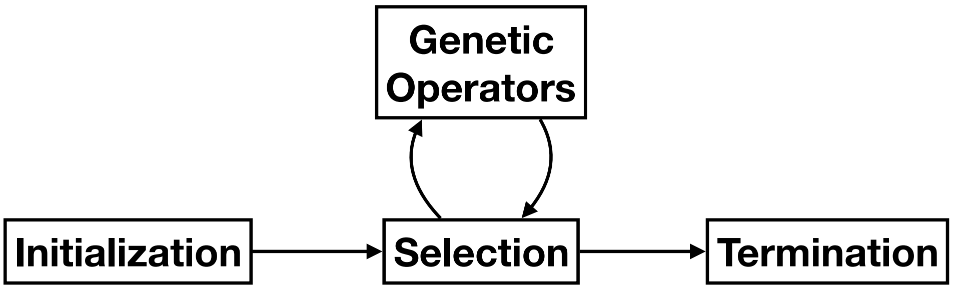


Figure 2.1: General evolutionary algorithm's main steps. The initialization steps produces a population of initial candidate solutions. The selection step selects the best individuals in a population. The genetic operators modify the current solution. The termination step assures that an EA finishes.

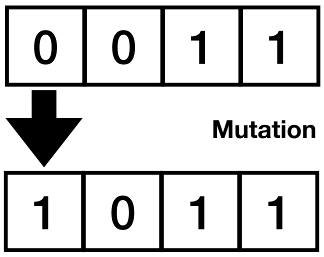


Figure 2.2: The mutation operator in an evolutionary algorithm. The first 0 in the vector representation mutates to 1.

Figure 2.1 shows the main steps of an evolutionary algorithm (EA) [23]: initialization, selection, genetic operators, and termination. Prior to starting an EA, we need to define a problem representation [21] that is suitable for manipulation in the steps of an EA. The representation is usually domain specific, such as a vector of values [139] or an abstract syntax tree as it is usual for genetic programming [167].

Initialization: the first step of an EA is the initialization that creates an initial population of candidate solutions, usually called *individuals*. The initialization can be done either randomly [150], or by using some domain knowledge [155]. The size of the initial population can also vary, depending on the characteristics of the problem. The initialization of an EA is important for the performance, as the initial population represents the pool from where we evolve the next solutions: having diverse individual is usually good as it allow us to better explore the search space [54].

Selection: an EA evolves the population across multiple generations. For each generation, an EA employs a *selection* stage that insures the survival of the best individuals to the next generation by the means of a fitness function. A fitness function takes as input an individual and outputs a numerical value to represent how fitted an individual is. The fitness function can be multi-objective [164, 321] when we want to optimize for multiple objective, such as run-time and energy consumption. When the fitness function is multi-objective we obtain a Pareto front [320] of solutions. The selection step computes the fitness for all the individuals in the population and further selects the population size top fittest individuals to survive to the next generation.

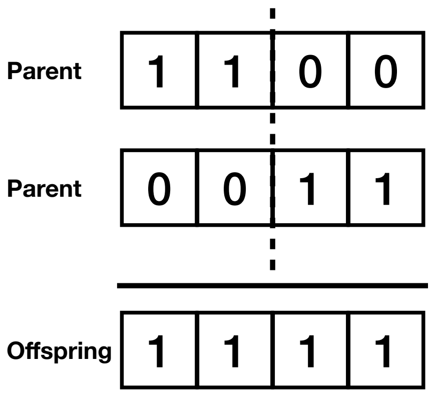


Figure 2.3: An example of crossover operator that takes half of the genes from the first parent and the other half from the second parent.

Genetic operators: the genetic operators modify individuals in the population in the hope of obtaining better fitted individuals than the original one. An EA applies the genetic operators over the individuals that survived the selection. The widely used genetic operators are: mutation, crossover, and reproduction. Mutation introduces new genetic material in the population by changing a small part of an individual. Figure 2.2 shows an example of the mutation operator, where the first *gene* in the individual changed from 0 to 1. The crossover operator combines two individuals together in one (or more) *offspring(s)* that have characteristics from both their parents. Figure 2.3 shows an example of an one-point crossover [316] between two individuals in the vector-based representation. The offspring takes the first half genes from the first parent and the second half from the second parent. Finally, reproduction replicates a fitted individual to survive in the next generation.

Termination: the process of selection and the application of the genetic operators repeats until a *termination* condition occurs. An EA might terminate for two reasons: either the EA has reached some fitness threshold, or the algorithm finished its resources (*i.e.* runtime). When termination occurs, an EA returns the fittest individual(s) that it saw as the solution.

Genetic Programming (GP) [167] is an evolutionary algorithm that evolves populations of computer programs. GP uses random mutations, crossover, a fitness function, and generaly uses a tree based representation to evolve programs aimed at solving a particular task.

Algorithm 1 shows the basic algorithm for genetic programming. The algorithm first initialize the initial population at line 1. The while loop between lines 2 – 5 runs until we reach a termination criterion, such as the number of generation. In each iteration of the loop, we process a generation: line 3 mutates the individuals in the population; line 4 applies the crossover operator; and line 5 selects the most fitted individuals to preserve them in the next generation. Finally, Algorithm 1 returns the final population.

The literature provides different GP approaches: Tree-based Genetic Programming [167], Stack-based Genetic Programming [52], Linear Genetic Programming (LGP) [49], Grammar-based Genetic Programming [217], Extended Compact Genetic Programming (ECGP) [276], Cartesian Genetic

Algorithm 1 Genetic programming basic algorithm.

```
1: $P \longleftarrow initialize()$
2: while $!termination()$ do
3:     $P \longleftarrow mutate(P)$
4:     $P \longleftarrow crossover(P)$
5:     $P \longleftarrow select(P)$
6: return $P$
```

Programming [227], Probabilistic Incremental Program Evolution (PIPE) [275], Strongly Typed Genetic Programming (STGP) [229], Genetic Improvement of Software for Multiple Objectives (GISMO) [175].

2.2 Search Based Software Engineering

Harman and Jones introduced the term Search Based Software Engineering in 2001 [126] to define a new approach to software engineering (SE) problems. They observed that a lot of SE problems ca be seen as optimisation problems, for which we can use search techniques. SBSE uses meta-heuristics [46] to navigate the search space of potential solutions until it finds an acceptable one. SBSE was used to solve SE problems in requirements engineering [25], software design [263], verification [110], maintenance [34], modularization [205], management [64], or testing [130]. A survey of SBSE [130] shows an exponential grow in SBSE work since its inception.

The idea of using search based techniques for software transplantation was proposed (but unimplemented and unevaluated) by Harman *et al.* in the keynote of the 2013 Working Conference on Reverse Engineering [129] (WCRE). In this thesis, we use search based software engineering techniques to implement the software transplantation problem, by transplanting non-trivial, and useful functionality, between popular, real world systems.

2.3 Genetic Improvement

One of the most successful search technique that SBSE uses is *Genetic Improvement*. Our automated software transplantation approach improves the functional properties of the *host* program and thus, is a form of genetic improvement. This section reviews the genetic improvement area and its successful applications in software engineering.

Genetic Improvement (GI) [247] uses automated search techniques to find versions of an input program that are improved for some given criterion. GI takes as input an existing program and an improvement criterion (*e.g.* energy consumption [51]). Further, GI creates variants of the input program by usually modifying its source code, but also its binary code [280] or parameters [342] with the aim of improving the given criterion over the input program. Recently, Yoo proposed embedding GI into the programming languages [349]. In his vision, GI would act as a compiler that takes as inputs human-written programs and outputs versions of the programs optimized for different objectives (*e.g.* energy consumption, runtime, *etc.*).

Petke *et al.* [247] surveyed GI papers between 1995 and 2015. They found a rapid increase in recent publications in GI: 60% of the GI papers were published between 2013 and 2015. Petke *et al.* consider as GI the approaches that:

- use metaheuristics search to explore the space of possible program variants;
- allow non-semantics-preserving program variants;
- start with an existing program as input for improvement;
- produce modified variants of the input program as output that is improved over the input program with respect to a given criterion.

The most popular search technique that GI uses is evolutionary computation. 96% of the core publications in GI use evolutionary algorithms, in particular genetic programming [247]. Our approach for automated software transplantation uses genetic programming as well, reviewed in Section 2.1.

Ada Augusta Lovelace was the first one to mention program optimization back in 1843 [199], even before electronic computers existed. Lovelace realized that there can be multiple valid syntactical variants of the same program that have the same semantics but exhibit different execution time. Out of these variants, we need to chose the optimal one, very similar with GI for execution time improvement [181]. In the history of the modern computation, Ryan *et al.* [326] are the first ones to implement genetic improvement in 1995. Their approach used genetic programming (GP) [167] to convert a sequential program into parallel variants to reduce the execution time.

In GI the improved program can be improved functionally (*e.g.* bug fixing [185], transplantation [30, 30]) but also non-functionally (to improve performance measures). For performance improvements, the semantics are intended to remain unchanged. In terms of the transplantation formalism that we will introduce in Section 4.3, GI does not use adapters, so $\alpha_i = \alpha_o = \varepsilon$. GI uses testing to reject candidates where $P_D \neq P_H \vee Q_D \neq Q_H$.

GI is an approach to program synthesis, which is a long-standing research challenge [207] at the interface between programming languages and software engineering research, that has recently been the subject of much activity, such as the Microsoft Flash Fill system [114]. Genetic Improvement (GI) uses existing code as 'genetic material' to automatically improve systems. GI was previously used very successfully, to improve the non-functional properties of real world systems and in fewer cases in functional improvements [185, 286]. Our approach to software transplantation builds on recent work in genetic improvement [17, 128, 178, 184, 237, 303, 338] and is the first work to use code transplants for transplanting non-trivial, useful, and real-world functionality between totally different software systems.

GI for non-functional improvements: the literature provides a lot of examples of GI improving the non-functional properties of real-world systems. The GISMOE challenge [128] proposes the use of GP for constructing a Pareto front [277] of programs, all of which share the functionality, but each of them differ in their non-functional trade-offs. Langdon *et al.* [178] showed that GI scales enough for processing a 5000 lines of code system. GI produces a version of Bowtie2 [181] that is, on average, 70 time faster than the original one, and is at least as good as the original one considering the functional proprieties. Gen-O-Fix [303] is a an embeddable framework for dynamic adaptive genetic improvement programming. It continually improves a software system running on the Java Virtual Machine, for different nonfunctional properties. White *et al.* [338] also improved the non-functional properties of a software systems by using genetic programming. Their approach takes as input the implementation of a function, and uses GP for evolving a semantically equivalent versions, optimised for a reduced execution time, under a given probability distribution of the inputs.

Genetic Improvement was also shown do be very effective for dramatically scale-up both benchmarks [237, 338], and real world systems. GI improved a 50k lines of code genome processor [178]. Sitthi-amorn *et al.* [293] used GP for automatically simplifying procedural shaders. Their framework computes a series of increasingly simplified shaders. The user can explore different trade-offs between speed and accuracy of the represented image. Their approach is applicable to multi-pass shaders and perceptual-based error metrics. Langdon *et al.* [177] successfully used GP for increasing the performance of domain specific software, written by specialised developers in that area. Specifically they improved the nVidia application `Stereo Camera` [1], developed by expert members of nVidia team. In this example, GP is enhanced by autotuning. Legacy GPGPU C code is optimised and refactored for modern parallel graphics hardware and software, such that it can take advantage of running on far more powerful hardware than on which it was designed to run. The speedups obtained are between 1.053 — 6.837 on different hardware configurations. Langdon *et al.* [180] obtained huge optimisations, of over 35% in some cases, a critical component of health-care industry software. They optimised `Nifty Reg` [2] across 6 different nVidia based GPUs. The optimised versions of `Nifty Reg` were tested by comparing the outputs with the ones of the original program (*i.e.* not week proxies [257]). The results showed that the difference in the output is always within the one caused by floating point accuracy. Langdon *et al.* [179] surveyed GI of general purpose computing on graphics cards, an important area in GI.

Loop perforation [138] is a related approach that transforms the loops such that them will execute only a subset of their original iteration. The goal in this case is to reduce the amount of computations required for obtaining the results. By reducing the amount of computation, the program's execution time, or power consumption are improved. By exploring the space of loop perforation in a program, the idea is to explore the trade-off between performance (execution time for example), and accuracy of the resulted program. By loop perforation, the program is loosing some accuracy (by removing some computations), but in some cases the accuracy lost is not significant, and is worth compared with

[1]`http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/StereoCamera_1_1.tar.gz`
[2]`http://sourceforge.net/projects/niftyreg/`

the performance improvement. Sidiroglou-Douskos *et al.* [289] used loop perforation for obtaining a Pareto front of trade-offs between the performance and accuracy, in the case of 7 benchmark programs. In their work, the authors compared the results of exhaustive search, with the ones of a search-based approach (a greedy algorithm). The greedy algorithm shown to be far faster (for example, in the case of `streamcluster`[3], the exhaustive search required 3941 minutes running time, while the greedy algorithm required only 31 minutes running time) than the exhaustive one, but with the trade-off of obtaining sometimes less than optimal points.

GI for functional improvements: previous work on GI has shown that it can also improve the functional properties of an existing program by repairing broken functionality. CodePhage [286] uses code transplants for automated bug fixing, work that we discuss in Section 2.13. Le Goues *et al.* [184] surveyed the application of genetic programming for automatic error repair. Arcuri *et al.* [17] proposed a co-evolutionary [121] approach to automate the task of fixing bugs. In their work, programs and test cases co-evolve by using a competitive co-evolution model: the programs are preys, while the test cases are predators. That is, program are evolved for passing the test cases, while the test cases are evolved for catching more bugs in the programs.

One of the most famous tools that fixes bugs by using genetic programming is `GenProg` [185]. `GenProg` is a generate-and-validate patch generation system that starts from the hypothesis that in a software system that contains a set of bugs, there might be code in different locations that may fix the bugs. Under this assumption, `GenProg` uses GP identifying the bug location, the code that would fix the bug, and for moving code. The reported results [111] are impressive: it fixes 55 out of 105 considered bugs. `AE` [333] is an optimised version of `GenProg` that reduce the patch search space by using a set of equivalence tests, for identifying equivalent patches. `AE` uses `GenProg`'s infrastructure, and benchmarks. Again, the results reported are impressive: it fixes 54 out of the 105 considered bugs in the case of `GenProg`.

However, Qi *et al.* [257] published a recent paper at the International Symposium on Software Testing and Analysis (ISSTA) 2015. Here, the authors explored in depth the patches resulted from some of the current approaches for fixing bugs, by using genetic programming. More concretely, they carefully analysed the results of `GenProg`, `AE`, and `RSRepair` [256] (an approach similar with `GenProg`'s one, but using random search rather than GP). In this paper, the authors discovered some problems with the evaluation of the automatically produced bug fixing patches. Their results shown that `GenProg` produced a correct patch for only 2 of the 105 considered defects; `RSRepair` produced a correct patch for only 2 of the 24 considered defects; while `AE` produced a correct patch for only 3 of the 105 considered defects.

There are two main reasons for these failures in evaluation: weak proxies, and functionality deletion. Weak proxies refers to evaluating the exit code of the patched program, rather than checking its output. Functionality deletion refers to the fact than in the analysed tools, in the vast majority of

[3]`http://parsec.cs.princeton.edu/`

```
 1  void sumProd(int[] a, int n){
 2      int i;
 3      int s = 0;
 4      int p = 1;
 5      for(i = 0; i < n; i++){
 6          s += a[i];
 7          p *= a[i];
 8      }
 9      write(s);
10      write(p);
11  }
```

(a) Example function to slice. This function computes the sum and the product of the elements in an array.

```
 1  void sumProd(int[] a, int n){
 2      int i;
 3      int s = 0;
 4
 5      for(i = 0; i < n; i++){
 6          s += a[i];
 7
 8      }
 9      write(s);
10
11  }
```

(b) The slice for the program in Figure 2.4a given the slicing criterion $C = (9, s)$.

Figure 2.4: Program slicing example.

the cases, the patch was just deleting functionality, without actually fixing any bug. The bug was not longer manifesting because the entire piece of code that contained the bug was removed. The regression tests failed to catch this problem because of the weak proxies.

In our work we carefully avoided using weak proxies as a final validation of the transplant. In fact, our multilingual transplantation approach take advantage of the weak proxies to better guide the search as we show in Section 5.4.4. All our test cases check the output of the program. We are not removing any functionality from the host, since the only thing that μSCALPEL or τSCALPEL might do in the host system, is to add the code of the organ. For a better evaluation of the transplants, we manually analyzed and tested all the beneficiaries of the transplants.

2.4 Program Slicing

In order to transplant code from a donor to an unrelated host, we must capture the code upon which the chosen functionality depends in the donor. This problem is very closely related to previous work on program slicing: our *organs* can be thought as slices of the donor programs. Mark Weiser introduced program slicing in his seminal 1979 doctoral thesis [335], creating an important line of research. Surveys on slicing techniques [39, 124, 312, 312], nomenclature [290] and empirical findings [43] can be found the literature, but are not particularly relevant for autotransplantation. Here we will briefly present the concepts from program slicing directly related to our work.

A *backward* program slice [337] S is constructed with respect to a slicing criterion which denotes the sub-computation of interest [335, 337]. A slicing criterion is: $C = (p, x)$, where p is a program point and x is a variable name. Informally, S consists of all the statements in a program P that may affect the value of x at the program point p. In our work, the slicing criterion is the human-generated annotation that capture the functionality to be transplanted: the entry point of the *organ*, together with the observational behaviour [42] of the host system, captured by the organ test suite.

Figure 2.4 shows an example slicing process. The program in Figure 2.4a computes the sum and the product of the elements in its input array a. Given the slicing criterion $C = (9, s)$, Figure 2.4b shows the corresponding program slice S that contains only the lines of code that *might* affect the value of the variable s at line 9.

Slices can be constructed in a backward (as the slice in Figure 2.4b) or forward direction [140]. While a backward slice [337] captures the code upon which the slicing criterion depends, a forward slice captures the code that is affected by the slicing criterion [38, 140]. In our work we used both concepts of backward and forward slice. The organ can be seen as a forward slice of the donor system, starting from the organ entry; while the vein can be seen as a backward slice of the donor's system, starting from the organ entry point, up to the entry point of the donor system (*i.e.* the '`main`' function).

Slices can be static [140, 337] (as the slice in Figure 2.4b), dynamic [165] and various flavours in between [59, 105, 322], including simultaneous dynamic slicing and union slicing [37, 119]. A static slice must respect the original program's behaviour with respect to the slicing criterion on all possible inputs, while a dynamic slice need only respect the program's behaviour for single input. Union and simultaneous dynamic slices respect the slicing criterion for a set of inputs.

Our approach is closer to dynamic slicing, since our approach, like other genetic programming based approaches, is guided by sets of test suites, but we use test suite observation rather than dependence analysis to determine data dependence relations and only a limited form of control dependence [101]. Our slices can therefore be thought of as a form of observation based backward slice [40, 42]. Binkley *et al.* [40] present ORBS [41], a tool capable of implementing observation based slicing. Observation-based slicing (ORBS) is a language-independent program slicing technique that is able to slice systems written in multiple languages. In their approach, a slice is obtained by repeated statement deletion. The results of a deletion operation, are validated by observing the behaviour of the program, at the program point annotated as the slicing criterion, by running a set of test cases.

As opposite to traditional program slicing techniques, our slices need only to capture the particular features of interest and not the entire computation on slicing criterion (the organ entry point). In this way, our approach is closely related to 'barrier' slicing techniques [169] and approaches to feature extraction [96, 123]. Krinke introduced 'barriers' [169] in program slicing. In his approach, *barriers* are nodes or edges in the program dependency graph [101] that are not allowed to be passed, during the computation of the program slice. The effects of introducing barriers, are in program slicing, to 'filter' the slices, and thus to present more relevant results to the user of the slicing tool. For autotransplantation, the 'barriers' block μSCALPEL or τSCALPEL to enter in sections of the donor program that are irrelevant for the computation of the feature of interest.

All previous work on program slicing constructs a slice as a subset of the original program with respect to the slicing criterion. By contrast, the 'slices' we require for transplantation are extracted from the donor, but have to be transformed (aka 'amorphous slicing' [122]) in order to make them executable within the host. The behaviour of the slices in our case, is not any more strict with the one of the initial program, but is defined by the organ test suite.

2.5 Code Synthesis

Multilingual Software Transplantation (Chapter 5) employs a form of code synthesis to both translate and transplant an organ simultaneously. *Code Synthesis* [117] aims at automatically finding programs in a target programming language that obeys some user intents. Our implementation of multilingual software transplantation (Chapter 5) currently uses genetic programming (Section 2.1) for code synthesis. The literature provides a couple of other alternatives that we review in this section. The other code synthesis approaches might be useful for software transplantation, considering that the holes that our approach needs to fill are usually very small in size (Section 5.3).

Areas such as programming languages, and artificial intelligence made a lot of contribution to code synthesis, one of the most important problem in the theory of programming [252]. The main idea in code synthesis is to construct programs that provably meet some specification by dividing them into sub-problems and composing the solutions to the sub-problems, idea explored in mathematics since 1932 [162]. Code synthesis approach usually use constraint solving to divide the top level program into sub-problems and to prove that the composition of the sub-problems entail the top level program [117]. The advances in automated theorem provers [103] made possible the advances in code synthesis. There are a couple of different approaches to code synthesis.

The enumerative approach: one of the approaches that the literature provides for code synthesis is the enumerative approach. The enumerative approach to code synthesis refers to enumerating the programs in the underlying search space by using counterexample-guided inductive synthesis (CEGIS) [116, 145, 295]. The basic enumerative search algorithm maintains a set of inputs, initially empty. In each iteration, it proposes a program that produces the correct output for all the inputs. Next, an enumerative search algorithms looks for a counterexample. If a counterexample is found, the algorithm appends it to the set of inputs and proceeds to the next iteration. Otherwise, the algorithm returns the proposed program as the final solution. Because usually the search space is very big, the naive enumerative approach does not scale. More advanced approaches use various techniques to prune the search space. Alur *et al.* [9] use a divide-and-conquer approach to restrict the search space by solving simpler problems whose combination solve the top level problem.

Esolver [8, 315] prune the search space by specifying as input a syntactic set of candidate implementations by a grammar. Only programs that this grammar specifies are considered in their synthesizing problem: syntax-guided synthesis problem (SyGuS). The SyGuS competition[4] evaluates different code synthesis engines. Sketch [294] introduced the idea of allowing a user to provide a partial program sketch that contains a skeleton and holes. Sketch automatically completes the holes given a specification. FlashFill [114, 115] is an programming by examples implementation of program synthesis for programs that process strings in spreadsheets. FlashFill is shipped with Microsoft Excel.

[4] `http://www.sygus.org`

The deductive approach: the deductive synthesis approach [112, 207, 254, 325] to code synthesis is a top-down approach [254] that recursively reduce the problem of synthesize a program p to synthesizing simpler sub-programs that when combined, entail p. The deductive approach views the code synthesis problem as a theorem proving problem for which it employs transformation rules, unification, and mathematical induction [206].

The constraint solving approach: the constraint solving approach [300] involves two stages. The first stage is constraint generation, where we try to generate a constraint whose solution is the program we try to synthesis starting from invariants [299], paths [298, 298] or input-output pairs [294]. The second stage is the constraint resolution stage, where the constraint solving-based solve the constraints from the first stage, by using an SMT solver. The disadvantages are about the constraints being to complex for the current SMT solvers to handle [55].

The statistical approach: statistical techniques such as machine learning [86, 224], genetic programming, MCMC sampling or probabilistic inference were also used for code synthesis. Machine learning is used in conjunction with other approaches for code synthesis [224] such as the enumerative approaches, or the deductive approaches to guide the search. Genetic programming [168] (Section 2.1) also synthesis code by using input – output pairs to specify the semantics of the code to be synthesized. Stoke [278] uses MCMC sampling [319] to synthesis optimized code, starting from an existing program, in a similar way with GI (Section 2.3).

Program synthesis [207] has recently been the subject of much renewed activity [117]. In terms of the transplantation formalism that we will introduce in Section 4.3, program synthesis starts with $\langle P_H \rangle \Box \langle Q_H \rangle$ and synthesises an O to fill the hole $\Box$ that makes the Hoare triple valid. Since code synthesis does not transplant any code, $P_D = P_H \wedge Q_D = Q_H$, $\alpha_i = \alpha_o = \varepsilon$, and $\mathscr{T}$ is unneeded. Harman *et al.* [127] and Jia *et al.* [149] graft new functionality (grown from scratch) into existing systems. This approach is like synthesis (growing from scratch) but uses test equivalence, more like GI, compared to traditional synthesis, which aims to be correct by construction.

2.6 Code Translation

Multilingual software transplantation (Chapter 5) involves code translation. A related topic to our multilingual transplantation approach is software porting [157] that modifies code to run in a different environment than the one for which it was designed to run. Porting can *transplant* a system to a new language [251] or to a new platform [283] (desktop or mobile device). Multilingual software transplantation implements both forms of software porting when transplanting Java written functionalities, for Android platforms, into Swift applications designed to run on iOS.

Porting to a different platform: porting a program on a different platform, such as porting an Android app on iOS, can be viewed as a form of code transplantation, into an empty host system, which is written for a different platform, potentially in a different programming language. GI was successfully used for porting a system to a new language and platform [176]. In their work, Langdon *et al.* used GI to automatically generate a parallel CUDA[5] kernel, with the same functionality as of the sequential version of gzip[6], written in C. For doing so, they start with a template kernel, provided by CUDA: `scan_naive_kernel.cu`[7]. This kernel is converted into a BNF grammar[8], which is evolved by the GI until the functionality of the original `gzip` system is reproduced. In this case the target language, CUDA, is similar with the source language C. Our automated multilingual transplantation work targets programming languages pairs whose syntax significantly differs such as Java and Swift.

Porting software on different platforms but in the same programming language is of a great concern in *High Performance Computing* [92], achieved either by designing the programs to run on multiple platforms or by using containers [36].

Porting to a different programming language: porting across different languages is source-to-source machine translation and is the most researched area in the code porting literature. The source-to-source machine translation approaches are inspired from natural language translation approaches that is a well researched area [91].

The existent natural language translation approaches are used for source-to-source translation as well, but the disadvantage is that they do not take into account inherent programming languages' structures captured by their grammar [152]. Not considering the grammar, leads the natural language SMT approaches to produce translations that are not grammatically correct and do not compile. The literature provides approaches that are particularly aimed at the source-to-source code translations [251].

Multilingual software transplantation entails the translation of organ from the donor into the host language. The literature provides three different approaches to both code translation and natural language translation: rule-based translation [282], statistical machine translation [160], or a combination of rule-based translation with statistical machine translation [291]. Existing work compares these approaches [311], concluding rule-based translation approaches are more reliable. The disadvantage of the rule-based systems are about lacking rules for constructs that need translations. Thurmair *et al.* [311] suggest hybrid approaches that initially use a rule-based translation system to translate the part of the input for which they have code, followed by statistical-machine translation for the parts of the input where the rules failed. This is similar with our approach in multilingual software transplantation (although we use GP rather than statistical machine translation) and with Simard *et al.*'s [291] approach.

Code translation is a difficult problem. The rule-based translation approaches maintain a set of transformations rules for elements in the source language to elements in the targets languages. Such approaches parse the input and apply the rules until either the entire input is translated, or until they

[5]http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html
[6]http://www.gzip.org
[7]https://github.com/pangeh01/gfwtcejorp/blob/master/lib/scan/scan_naive_kernel.cu
[8]http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf

reach constructs in the input for which the rule-based translation systems lack the translation rule. This process is similar with the way in which a compiler compiles the source code to binary code [2]. Tansey *et al.* [309] used a rule-based translation system to translate Java code to annotated Java code for more than 80k lines of codes, across four open source java projects. Trudel *et al.* [314] translate C code to Eiffel.

The compilers use translation rules to translate the source code to binary code, intermediate language, or a different source code in same cases. LLVM [183] first translates the source code into LLVM bitcode for optimization. LLVM can also translate from one of its supported languages (C, Ada, C++, Fortran, Objective C, Objective C++, and Java) to C or C++[9]. The ROSE compiler infrastructure [259] (Section 2.11.1) can be used to implement source-to-source translators between any languages that it supports (C, C++, Fortran, Python, or Java).

Rule-based translation works well for those parts of the source and target languages that change infrequently, like assignment statements and the API of the language's core libraries. Domain- or project- specific libraries are their Achilles heel, because of their number, diversity, and propensity to change and evolve. Keeping pace with these libraries is expensive [87]. Even if we don't consider the frequent changes in APIs, usually the number of APIs in real world programming language is too big to construct translation rules for each of them, as a rule-based translation system would need. The strongest evidence that this cost is prohibitive is the fact that no rule-based translator currently exists that translates arbitrary programs between mainstream high-level languages. The existing rule-based translators, such as `patniemeyer/j2swift`[10] or `eyob/j2swift`[11], usually support and translate only parts of the full specifications of the source language (*e.g.* they do not translate API calls).

Phrase-Based statistical machine translation [161] (SMT) approaches are a very popular translation approach in the case of natural language translation. Google translate uses such an approach for example[12]. The SMT approaches use machine learning to train a model with examples of translations from the source language to the target language. Karaivanov *et al.* [152] start with a natural language oriented SMT and extend it with programming language grammatical knowledge to avoid producing invalid programs. In SMT the translation process works by building up prefixes until the entire source text (or code) is translated. To take the grammar into account, Karaivanov *et al.* check the prefixes at each step whether or not them can be completed in the target grammar. If not, they discard the prefix. Taking the grammar into consideration, increases the compilation rate from 57% to 68%.

Statistical machine translation is promising, but depends on a large training corpus of aligned programs: the same functionality written in two programming languages, with an invertible function mapping equivalent functions and statements to each other. Such corpus is difficult to find for an arbitrary language pair. Our multilingual software transplantation approach uses rule-based translation

[9]https://llvm.org/devmtg/2013-04/krzikalla-slides.pdf
[10]`https://github.com/patniemeyer/j2swift`
[11]`https://github.com/eyob--/j2swift`
[12]`https://blog.statsbot.co/machine-learning-translation-96f0ed8f19e4`

for core language constructs, but grows code to fill the holes using approaches based on Genetic Programming (Section 5.4), so is more reminiscent of GI.

There are other technical challenges to overcome to integrating a statistical machine translation approach with transplantation. Statistical machine translation approaches require a large training corpus, *e.g.* Karaivanov [152] used $20,000$ examples of manual translated functions, as the training set for Java to C# code translations. The fact that we often most want to use translation to target new languages, like Swift, for which such aligned corpora are rare, makes this problem particularly acute. Clearly, one avenue for future work on SMT-transplantation hybrids will be to better leverage small training sets. Second, the success rate of the current state of the art is low: after a manual analysis, Karaivanov [152] identified that only about 30% of the translated functions are semantically equivalent to origin function. Practical MST requires a much higher success rate.

Section 5.5.5 reports our attempts to use both of the existing code translation approaches for transplantation: rule-based or SMT. Unfortunately, none of them worked: we couldn't found existing rule-based translation systems for our language pairs; and the statistical machine translation tools that we tried do not produce compilable code.

2.7 Code Search

Our automated software transplantation approach assumes the existence of a donor program and of an organ entry point in it. Further, we use context-insensitive program slicing on the call graph of the donor program in conjunction with GP, to search for the code of the organ. The literature provides a couple of alternative code search approaches that might remove the burden from the user of autotransplantation of identifying a suitable donor program.

A lot of code search engines allow the user to specify the code search query in textual forms. Code search engine such as Ohloh[13], Krugle[14], or Sando [285] return code snippets containing some keywords, or as specified by regular expressions. Web search engines, such as Google[15] can also be used to search for code by specifying keywords. Sourcerer [198] is a framework for automated code search at large scale, that supports keyword-based or structure-based search [108] (*i.e.* syntactic patterns in code, such as try – catch constructs).

Sniff [65] allows a user to specify a search code query in a textual format (*e.g.* "execute command"). Sniff extends the previous text based search approaches, by automatically annotating the user code with the documentation of the Java API's that the user code uses. Further, Sniff cluster the found code snippets according to their textual similarities and ranks the clusters according to their sizes by preferring the bigger ones. CodeHow [203] takes an user query written in natural language. Similar with Sniff, CodeHow uses the API documentation to enhance the search in the user code. CodeHow is implemented as a Microsoft Azure service. Its front-end is a Visual Studio extension.

[13] `https://code.ohloh.net/`
[14] `http://www.krugle.com/`
[15] `www.google.com`

All these approaches look only into the syntax of the source code. In automated software transplantation we are interested about the semantics of the code. Autotransplantation also does not have access to the syntax of the missing organ, only to an (under)approximation of its semantics, in the form of test cases.

There are code search engines that take a code fragment as an input query and return similar code fragments from a database of programs, such as Xsnippet [274], ParseWeb [310]. Code completion tools such as GraPacc [231] or Bruch *et al.*'s tool [53], similarly match an existent incomplete code snippet, with similar snippets in a code base, to identify code completion suggestions. The code completion approaches might be helpful in filling an *incomplete* organ, when our autotransplantation approach fails.

Facoy [159] is a code search engine that follows a code-to-code approach: finding code fragments that might be semantically similar to the input code, similar to type 4 code clones [35]. When receiving a code fragment, FaCoY auto-generates a query from it. Further, FaCoY searches for relevant Q&A posts that contain the most syntactically similar code snippets with the input one. Further, FaCoY collects natural language descriptive terms from the associated questions that it uses to find other relevant posts. From all the code snippets found in the answers to the relevant Q&A posts, FaCoY generates code queries, alternative to the one obtained from the user input. Finally, FaCoY returns the code fragments, from a repository, that match the generated queries.

Such approches that takes as input an existing code snippet and returns similar one might help autotransplantation to explore software transplantation for different implementations of the same organ: once an initial organ is transplanted, autotransplantation might use code-to-code search engines to find different implementations of the organ that might have different non-functional properties (*e.g.* run-time, energy consumption).

S6 [269] is a semantic code search engine that has the advantage compared with the previous approaches, of looking into the semantics of the code rather than only on their syntax. S6 allows the user to specify code search queries as keywords, class / method signatures, test cases, contracts, and security constraints.

Test-Driven Code Search (TDCS) [187] is an approach for code search that allows the user to provide the query in the form of test cases. This code search approach is probably the most feasible one for our autotransplantation approach, as we also specify the desired behaviour of the organ as a test suite. CodeGenie [187] uses the test cases both for textual search (*e.g.* use the name of the methods in the test cases as keywords in the search query) and for matching the output of a code snippet with the desired output, as specified by the test case. Lemos *et al.* extended CodeGenie [186, 188] to use Interface-Driven Code Search (IDCS) together with TDCS. IDCS extends the text-based code search engine by allowing the user to specify the interface definition of the searched function (*i.e.* its parameters and return type). This approach might be useful for automated software transplantation, as it uses exactly the inputs that define our organ: the organ's interface definition as defined by its implantation point in the host, and the organ test suite.

2.8 Maintainability Of Code Transplants

Automated software transplantation implies moving an organ from a donor program into a host program. A problem that naturally arises here is about maintaining the organ in-sync with its implementation in the donor. Ideally, we would like the transplant to take advantage of changes such as bug fixing, optimization, or functionality addition that the developers of the donor system do. Here, we can either redo the transplant when the organ in the donor evolves, or apply the changes in the organ in the donor to the organ that we transplanted in the host. Maintaining the organ in-sync with its implementation in the donor is outside the scope of this theses, but the literature provides some example of efforts to maintain code that was manually transplanted from a donor system, into a host one [222, 223, 265, 267]. None of these works are aiming for the automated software transplantation, but tackle the situations arisen after the manual transplantation took place and can be helpful for improving the organs' maintainability.

Meng *et al.* [222] try to automatically apply an edit operation, manually learned from the user, into similar contexts. They present SYDIT [221], a program transformation tool. SYDIT starts from an example edit operation, provided by the user, and generates a context-aware abstract edit script, which might be applied in similar contexts in the program. They identify similar contexts in the program, according to data and control dependencies. LASE [142, 223] is an extension of SYDIT tool, capable of learning from multiple examples, and automatically identify some target locations for applying the edit operation. As observed in the case of SYDIT, learning from just one example is often not enough, because the context identification step might cause too many false positive, when used only one example. The authors tries to fix the drawbacks identified for SYDIT, by implementing LASE.

Ray *et al.* [265] presented a case study of cross-system porting in forked projects. Forked projects refers to creating a new version of a software system by copying and modifying an existing one. The problem that arise here, is maintaining all the forked variants of a software system, when a new feature appeared or when a bug was fixed in one of the variants. In this cases, we have to port (a.k.a. *transplant*) the patch from the variant that implements the new feature, or fixes a bug. REPERTOIRE [267] is a tool capable of identifying ported patches (aka *transplants*) between different variants. REPERTOIRE takes as input the versions histories, and then identifies all the edits that were ported, by comparing the content of the diff files. It presents the user a sort of details about the port, such as time, location, authors, and time required for the port.

The literature provide some approaches that identify when a clone needs refactoring, according to changes to its clone peers. Considering that usually an organ is the clone of its implementation from the donor, such approaches could potentially recommend when an organ needs refactoring, according to the refactorings of its implementation in the donor. Balazinska *et al.*'s [26, 27] approach first classifies the differences in clone peers in 18 different categories (*e.g.* "Superficial changes", "Called methods"). Next, they rank the clones that need refactoring according to the categories of differences observed between the clone pairs.

CREC [351] is a learning-based tool that first automatically extracts clone groups that it divides in refactored and not refactored. CREC further uses these clone groups as a training set. Further, CREC extracts a set of 34 features related to the evolution behavior of the individual clones as well as to the relationships between changes is clone peers. Using these features and training set, CREC trains a classifier that recommends which clones should be refactored. Wang *et al.* [328] follow a similar approach, but with a reduced features set. CREC improves over Wang *et al.*'s tool's effectiveness.

Ray *et al.* also investigate the problem of keeping a transplanted organ in sync with its clones, as the organs change over time [266]. In their setting, an organ is manually moved to a host. When both the original organ and its clone change, the SPA tool checks the two patches for consistency, and flag the inconsistencies if found. In our work we automate the transplantation of functionality from a donor to a host.

Code reduce related approaches [268] could be helpful for improving the maintainability of our code transplants. Regehr *et al.*'s tool, C-Reduce, is capable of reducing a C program to a smaller program that preserves some behaviour of interest specified through test cases. Their main goal was to help users in reporting compiler bugs by obtaining a small test case that trigger the bug, from a real C program. This work could be applied in the case of our monolingual software transplantation approach (Chapter 4) as our approach already relies on a test suite to specify the desired behaviour of the transplant. After applying C-Reduce, we would obtain a minimal transplant that implements the functionality of interest and thus, it will reduce the transplantation maintainability burden as the reduced transplant will add less code for the developers of the host system to maintain.

2.9 Automated Fault Finding

There are a several approaches that generate crashing test inputs for Android apps. SAPFIX is a general automated bug fixing technique that can be used in conjunction with any of these approaches: given a test case that one of these approaches produces, SAPFIX tries to generate patches for the bug that the failing test case reveals.

Currently, SAPFIX uses Sapienz [6, 210] and Infer [56], that we discuss in Chapter 3, but could use any other test generation techniques that would scale at the size of Facebook's code base.

AndroidRipper [10] is an automated test generation technique that tests Android applications through their Graphical User Interfaces (GUI). AndroidRipper is based on GUI ripping [220] that is a technique that automatically traverses the application's under test GUI by opening all its windows and extracting all their widgets. Traditionally, GUI ripping was used to create a model of the application [220]. AndroidRipper uses GUI ripping to generate test cases during the exploration process. The authors tested AndroidRipper on “Wordpress for Android”[16] where it detected it detected four runtime crashes.

SAPFIX could also use concolic testing [55] systems for Android, such as ACTEVE [12] or Collider [143]. These approaches try to generate GUI events that execute once each widget on a given Android displayed screen. They do so by symbolically tracking tracking events from the point where they originate to the point where the Android application handles them. For each such event, ACTEVE or Collider track a symbolic constraint that uniquely identifies each concrete event that are handled in the same way. This allows them to record only one event for each possible handler.

A^3E [22] is a static data flow test tool that uses a taint-style data flow analysis on the applications' bytecode to construct a high-level control flow graph that captures legal transitions between the different screens. Further, A^3E uses this graph to explore them by retrieving the GUI elements and exercising them. A^3E uses two different exploration strategies to exercise the GUI elements: targeted and depth-first. The authors tested A^3E on a set of 28 popular Android apps on which they obtained a mean activity coverage of 59.39% and a mean method coverage of 36.46% for the depth-first strategy; and a mean activity coverage of 64.11% and a mean method coverage of 29.53% for the targeted exploration strategy.

SAPFIX could also use Dynodroid [204], a feedback directed random test tool or Android Monkey, which are other popular Android testing tools. Dynodroid views the application under test as an event-driven program that interacts with its environment through different sequences of GUI events [204]. Dynodroid instruments the application to first identify which events are relevant for the current state of the application. Next, it selects one of the relevant events by using a random algorithm that penalizes the widgets that were previously selected more frequently. Finally, Dynodroid executes the events. The authors evaluated Dynodroid on a set of 50 open source apps from F-Droid [17] and the top 1.000 top free apps from Google Play. Dynodroid found 15 unique crashes across all these applications. Android Monkey[18] is the default testing tool for Android applications shipped with Android and that uses random search to find sequences of GUI events that cause crashes in the application under test. Although Sapienz has previously been shown to outperform both [210], these earlier results also indicated that the three techniques are, nevertheless, complementary.

There are also model based test tools such as FSMdroid [302] and fuzzers such as Fuzzdroid [264] that could be used to complement our results. It is likely that *any* or all of the techniques listed above (and many more) might find additional faults not found by Sapienz and could thereby complement our initial deployment of SAPFIX. More generally, any static or dynamic analysis tool that scales to tens of millions of lines of code and 100k+ commits per week could be used as either a replacement or a complement to our use of Sapienz and/or Infer.

[16]http://android.wordpress.org/
[17]https://f-droid.org/en/
[18]https://developer.android.com/studio/test/monkey

2.10 Grow and Serve

'*Grow and Serve*' [149] is a GI approach related to software transplantation, where the contributions of this thesis proved to be useful. Grow and serve is an approach for automatically generating new functionality, starting from a set of user provided '*hints*'. The '*Grow and Server*' approach is an automated transplantation approach from scratch: as opposite to the software transplantation approach described in this thesis, in grow and serve we do not have a donor program, only a host one. GI automatically generates new, genuine, and useful functionality from scratch, starting from a set of user provided hints.

Jia *et al.* used the Grow and Serve approach for automatically growing Django[19] citation services using search based techniques. The grown service is able to automatically query an academic web server (google scholar[20] for example) for the citation information for a requested paper, and return the number of citations as an image or as an string. This service can be used by any existing Python based web application, for incorporating the citation count feature. In the first 24 hours of deployment on GP bibliography[21], the citation service was accessed 369 times from 29 countries.

The '*Grow and Serve*' approach is similar with the '*Grow and Graft*' approach [127], where the *graft* step is missing. Our GP system takes as input a grammar file, a test suite for the desired feature, and a set of '*hints*'. The output is a program that passes all the test cases in the test suite.

The grammar file specifies data type and APIs that might be useful for growing from scratch the desired feature, but not all them necessary required. In our experiments, the suggested APIs contained: handling HTTP requests, parsing HTML, string and list manipulation, along other APIs. Thus, the evolved program, contains a sequence of assignments, and function calls to the APIs in the grammar file.

The rest of human guidance is provided to Jia *et al.*'s approach by the means of the fitness functions. They used 12 fitness components, merely hints provided by the user, for guiding the search process. These fitness components can be divided into 3 categories: Inclusion(I) — hints about what APIs from the grammar file should be included; Ordering(O) — hints about the order in which different APIs should be called; and Necessary(N) — hints about some APIs receiving correct input, or returning correct output. Essential(E) fitness components are the one implicitly available: the program should compile, and should not crash.

In their experiments they evaluated the results of different fitness components combinations. When using all the fitness components (ENIO), 16 out of 30 tries successfully growth the functionality. When using ENO, 9 out of 30 tries successfully growth functionality; while when using EIO only 1 out of 30 tries was successful. In all the other cases(ENI, EN, EO, EI, and E), they do not report successful grown functionality.

[19] `https://www.djangoproject.com`
[20] `http://scholar.google.com`
[21] `http://www.cs.bham.ac.uk/~wbl/biblio/`

2.11 Tools

The research reported in this thesis is build on state of the art industrial tools. This sections provides a brief overview of the main tools that we used in the automated program repair and automated software transplantation work, specifying which part of the reported research used each of the tool .

2.11.1 ROSE Compiler Infrastructure

ROSE [22] is a compiler technology developed at Lawrence Livermore National Laboratory[23] (LLNL). Our tool for monolingual software transplantation uses ROSE as described in Section 4.5.2. In monolingual software transplantation we used the ROSE compiler infrastructure to enable our monolingual transplantation tool to read source code, do the necessary transformations, and output the modified source code. ROSE provides us powerful APIs for manipulating the Abstract Syntax Tree[24] (AST) of the programs, and quickly do the necessary source code transformations.

The most part of its source code is open source, excepting the C++ frontend. ROSE is using EDG[25] as a frontend, which is not open source, and thus the EDG together with the ROSE specific part for connecting to EDG are closed source, and provided as a binary. However the EDG source code might be obtained under an academic license.

ROSE is used for building source-to-source program transformation and analysis tools for large-scale real world systems. ROSE is currently supporting a set of programming languages: C(C89, C98), C++(C++98, C++11), UPC, Fortran, OpenMP, Java, Python and PHP. This characteristics make it well suitable for developing our monolingual software transplantation tool: μSCALPEL (Section 4.5), since we require source-to-source transformations. This section provides an overview of ROSE compiler, motivates its usage in our tools, and provides insights into how we use ROSE.

According to the developers of ROSE [259], ROSE aims to be a library (and set of associated tools) to quickly and easily apply compiler techniques to source code, for improving performance and productivity. Secondly, ROSE aims to be a research and development compiler infrastructure that allows the users to write own custom source-to-source translators, to perform source code transformations, analysis, and optimisations. This second characteristic of ROSE makes it well suited for automated software transplantation. ROSE showed to be very scalable, and capable of handling real world systems, without any problems.

The literature reports a lot of applications of ROSE, on real world systems. ROSE was successfully used for: the verification of the one-definition rule in C++, scaled on whole-program analysis [260]; generating message perturbation for improving the testing of distributed memory applications [324]; inspecting the code quality of HPC[26] applications [243]; analyzing and visualizing software architectures for the entire program [242]; allowing the specification of bug patterns [261];

[22]http://rosecompiler.org
[23]https://www.llnl.gov
[24]https://en.wikipedia.org/wiki/Abstract_syntax_tree
[25]https://www.edg.com/index.php
[26]https://en.wikipedia.org/wiki/Supercomputer

showing a unified single-view visualization of the entire program software architecture [240]; the security analysis of source and binary coe, for shared and distributed memory [258]; distributed memory analysis [216]; signature visualization of software binaries [239]; automatic parallelization for multicore systems [194]; software quality analysis of binaries [241]; clone detection in binary applications [273]; source-to-source outlining for whole program optimisations [193]; automatic generation of an abstraction-friendly programming model [191]; semantic-aware automatic parallelization [195]; implementing a OpenMP 3.0 research compiler [192]; automatic detection of c-style errors in UPC[27] code [250]; transformations exploiting array syntax in Fortran programs [297]; automatically optimising hardware-software codesign [284]; automatically translating MPI[28] applications to a data-driven form [234]; identifying the impact of application-level optimizations on the energy consumption of [262]; auto-scoping for OpenMP[29] tasks [271]; exascale fault-tolerance research [197]; enabling concurrency by integrating constraint programming [45]; implementing HOMP [196]; using constants loop bounds in finite state machine, for verifying polyhedral optimisations [279]; supporting multiple accelerators in high-level programming models [344].

Very helpful in our work are the AST transformations capabilities of ROSE. Translators generated by using ROSE, first read the input source code, generate the AST of the source code, allows parsing and transformations of AST, and finally outputs the code resulted after the transformations.

The architecture of ROSE is similar to other compiler architectures. The frontend transforms the input source code into a language independent form, the AST of the program. ROSE has built-in support for merging the ASTs of all the source files in a project, which enables us to do whole-program analysis, by using ROSE. The midend contains a powerful API for AST based program transformations and analysis. The midend is capable of generating back the source code, from the potentially modified AST, by keeping all the preprocessor directives and comments. The code resulted by the AST is very similar with the original one, in the absence of the annotations. Thus ROSE generated code is readable and maintainable. The last layer of ROSE is the the backend. In the backend, ROSE is just calling a vendor compiler (GCC for example) for generating the binary code out of the source code (the original one, or the transformed one).

Figure 2.5 shows all the steps involving in processing one (or more) source file(s) with a ROSE based translator. ROSE takes as input one or more source files, written in any of the supported languages. First we start by calling the frontend for the language in which the source file under compilation is written. Then, ROSE is transforming the ASTs specific for the frontend, into an independent AST, parsable and modifiable by using the ROSE AST traversal and transformations APIs. After the code is recognised, we can use the AST traversal and AST Rewrite APIs, and ROSE automatically apply the desired transformations on the input source code. Next, the code generator is

[27]http://upc.lbl.gov/
[28]https://en.wikipedia.org/?title=Message_Passing_Interface
[29]http://openmp.org/wp

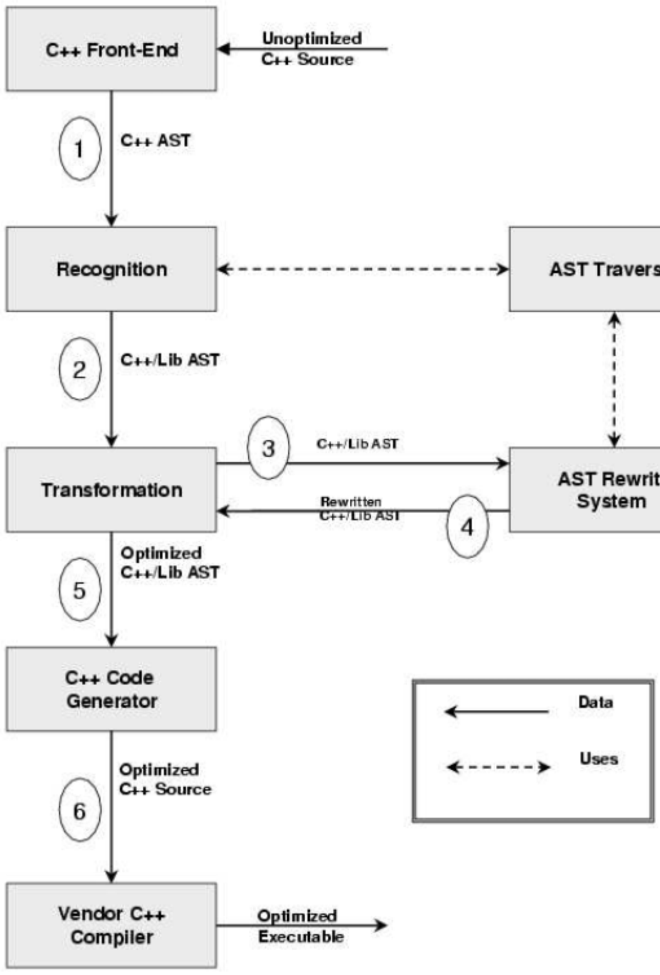


Figure 2.5: The processing of input source code with ROSE based translators.

called, for generating the modified code according to the rewritten AST. Lastly, ROSE call the vendor compiler, for generating the binary code.

In ROSE, the AST nodes are represented as internal representation (IR) nodes. For being capable of generating back the source code from the IR graph, these nodes contain more information than other compilers traditionally include in the ASTs nodes. Thus, the general graph composed by the IR Nodes, and the links between them, is a decorated AST. It also contain informations about: types, symbols, compiler generated IR nodes, supporting IR nodes (source file information nodes for example), and attributes. The attributes can be added by the user of the ROSE API to any node in the AST. Thus, it is possible to annotate nodes for further transformations, or to do different analysis, according to the contexts of a node, temporary saved in attributes. We can consider the traditional AST of a program, to be a subgraph of the graph composed by the all IR nodes available for an input program. ROSE keeps so much informations in the IR nodes for being able to regenerate the original source code, as close to the original as possible (when no transformations are involved).

From ROSE, we are (the) most interested in the AST manipulation mechanisms. These mechanisms are divided into 3 main categories: AST Query Mechanisms, AST Traversal Mechanisms, and AST Rewrite Mechanisms.

AST Node Queries represent a very simple and easy to use mechanism for obtaining information from the AST. It is also very efficient, by not requiring an explicit traversal of the AST. In practice, the node queries are only a single function call, that returns all the nodes in the IR graph, that match the query criterion. Queries can be also combined, such that we obtain composite queries, for a better filtering of the results. The builtin queries implemented by ROSE are: IR nodes, string, or number queries. Another advantage of the query mechanisms is that the user is allowed to call a custom defined function, for each node returned by the query. Like this we can do program transformations very easily. For example, we can replace automatically a variable name with an annotation for variable renaming in software transplantation Section 4.4.

The most important query mechanism is represented by the IR nodes queries. These can be of 3 kinds: queries of subtrees of the AST, starting from an SgNode (the class to represent an AST node in ROSE); queries of a node lists that returns all the nodes in the AST that have the desired characteristic; and queries of memory pools, which have the advantage to be extremely fast, by not requiring at all the traversal of ASTs. However the queries of the memory pool, do not contain any context information for a node, since we are just returning an area of heap memory, containing all the nodes of a kind (this is possible because ROSE keeps clearly separated in the memory, all the nodes of different kinds in the ASTs).

AST traversal mechanism is a very simple to use API for traversing the nodes in the program's AST. ROSE has capabilities for limiting the deep of the traversal. We can traverse the entire AST of the input program (including the nodes from the external included files), just the nodes in the input files for the translator, or just the nodes in a certain input files. The traversals can be in preorder, postorder[30], or a combination of these preorder and postorder, by doing both of them in the same time.

At the traversal, ROSE calls the function '`visit()`' for every node in the AST. According to the order in which this function is called, the traversals can be: simple processing, when the '`visit()`' function is called at every node in preorder or postorder, without using the attribute mechanisms; prepost processing, when the '`visit()`' function is called both preorder(before the child nodes are visited), and postorder(after the child nodes are visited), again without using the attribute mechanism; topdown processing, which is a preorder traversal of the AST nodes, with a restricted form of inherited attributes: only one attribute of a parent node is inherited by all its children; bottomup processing, when the synthesized attributes are used: here we compute from a list of synthesized attributes of children, the attribute to result for the parent node; topdownbottomup processing, when both the

[30]`https://en.wikipedia.org/wiki/Tree_traversal`

capabilities of the previous 2 kinds of processing are used: it computes both the inherited and the synthesised attributes, for children and parents.

The AST Rewrite Mechanism provides the APIs for the actual program transformations. At every node in the AST we can use this mechanism for rewriting the node, removing the node, or appending a new node. ROSE implements 4 different levels of the rewrite mechanism, all of them providing insert, replace, and delete operations. Level 1 contains the most low level API, where we are actually working on the Sage III [44] interface, by modifying the individual IR nodes. This level is used for implementing the higher level ROSE rewrite APIs. In Level 2 we still operate on the nodes in the AST, being similar with the Level 1. However, for Level 2 rewrites, the operations are performed on any statement and not on the containers that store the lists of statements. The required modifications on these containers, are internally implemented by ROSE, using the Level 1 interface.

Level 3 is a mid level interface. As opposite to the previous one, here we operate on strings rather than on AST nodes. This interface is build on the top of Level 1 and Level 2, and allows a very easy manipulation of the AST, by specifying string level modifications. In this case the operations are immediate and local on the specified node of the AST. Level 4 is a high level interface, which operates also on strings. This is the most easy to use interface, and it must be used within a traversal. It has support of relative positioning of the string level specified nodes, within the AST. The position is relative to the current not that is to be processed.

Even if the Level 4 or Level 3 are the most easy to be used, by automatically generating the correspondent AST structure from the provided string, there are some problems with them. These levels are not very robust, and a lot of errors appears when using them in complex contexts. Because of this, in the implementation of μSCALPEL (Section 4.5) we used in generally the Level 2, low level layer.

From our experience, ROSE proved to be a very scalable and robust program transformation API, by benefiting from the use of compiler technology. Its advantages can be resumed as as presented in this section makes it very suitable for the use in the implementation of μSCALPEL (Section 4.5).

2.11.2 Docker

Docker [88] is a recent technology used by developers or system administrator for building, shipping, and running applications in an isolated environment, and operating system independent. Thus, docker allow us to distribute all our applications very easy, avoiding for the users of our tools all the process of compiling our tools, and the dependencies of them. Especially because we use ROSE [31] (Section 2.11.1) compiler infrastructure, making the installation process of our tools easy is very important. ROSE compiler infrastructure might take even 10 hours to be compiled, while all its dependencies are hard to be met, and time consuming. By using Docker containers, all that the users of our tools must do is to run to scripts. The Docker container then automatically set all the running

[31] http://rosecompiler.org

environment, and start the run process of our tools. So, the use of docker makes the ROSE Compiler much easier to deploy and get started with. We have dockerized μSCALPEL. μSCALPEL is public available[32].

Compared to virtual machines, docker is more lightweight and efficient, since it provides an additional layer of abstraction of operating-system-level virtualization. As opposite to the virtual machines, the docker containers are not abstracting the hardware components of an operating system, and thus are must faster. Also, the layered architecture of containers makes it space efficient, by using any installed components from different containers, where available and proper to do so.

More than just providing an easy installation process, and isolated run for not allowing the environment of our users to affect the running of our tools, Docker also provides an additional layer of security [89]. So Docker is also protecting the users of Docker containers, by any potential malicious behaviour of the tools to run in the container. The environment in the container is isolated, and it might not affect in any way the host operating system.

2.11.3 Sapienz

SAPFIX (Chapter 3) uses Sapienz to identify candidate crashes that require fixing and to check that fixes pass the original failing test(s), as well as generating new tests and a partial approach to detecting some categories of knock-on issues that the candidate might introduce.

This section presents an overview of the Sapienz and its deployment at Facebook, the full technical details of which can be found elsewhere [6].

Sapienz uses multi-objective Search Based Software Engineering (SBSE) [120] to automatically design system level test cases for mobile apps [210]. Sapienz tries to automatically explore and optimise test sequences that are aimed at find crashes. The multi objective criterion of Sapienz are to minimize the lengths of the test sequences, while simultaneously maximising coverage and crash finding capabilities. In order to do so, Sapienz combines random testing with search based exploration.

In the paper that proposed Sapienz [210] the authors report promising results. Sapienz significantly outperformed the state-of-the-art at the time of writing Dynodroid tool [204] and Android Monkey [13] in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length.

Further, the authors applied Sapienz to the top 1,000 Google Play apps. Here Sapienz found 558 unique new crashes that were previously unknown to the developers. The authors contacted developers for 27 crashing apps. 14 of these developers confirmed that the crashes are real and six of them fixed those crashes.

Before starting its execution, Sapienz instruments the application under test. Sapienz can use white box, grey box, or black box instrumentation. When Sapienz has access to the app's source code,

[32]http://crest.cs.ucl.ac.uk/autotransplantation/downloads/artifact.html

it instruments the app statements. When Sapienz only has access to the APK as usually happens for the real-word applications, Sapienz instruments the app at the method level (grey box). Finally, when the app cannot be repacked as is often the case for commercial applications, Sapienz treats the app as a black-box.

To seed the text inputs for the app Sapienz extracts the string constants statically defined in the app code. Further, Sapienz uses these constants as seed for generating realistic strings to inject into the app which has been found to be useful [7, 48, 106]. The MotifCore component of Sapienz [210] combines random fuzzing with search based software engineering to generate test sequences.

The GP algorithm that MotifCore uses has two types of genes: low level atomic genes and high-level motif genes. The low level genes correspond to individual adb events, such as clicking on a button. The motif genes corresponds to more complex actions, such as executing login in the app under test.

Sapienz uses random search to initialize the population in MotifCore. The TestReplayer component of Sapienz has the purpose to evaluate the individual fitnesses during the genetic evolution process. Individuals in the GP process are test scripts that corresponds to executable Android events that can be executed through the Android Debugging Bridge (ADB). Finally, at the end of a Sapienz run, it produces a set of Pareto-optimal individuals and test reports.

After publishing the Sapienz paper, the authors launched Majicke [98], a London based start up in 2017. Facebook acquired Majicke and Sapienz is now fully integrated in the Facebook infrastructure, and is a core component of Facebook testing for mobile apps, for which it finds 100s of crashes every week, approximately 75% of which are fixed by developers [6]. Sapienz augments traditional Search Based Software Testing (SBST) [125, 218], with systematic testing. It is also designed to support crowd-based testing [209] to enhance the search with 'motif genes' (patterns of behaviour pre-defined by the tester and/or harvested from user journeys through a system under test [211]). When it detects a crashing input sequence, Sapienz reports a stack trace, together with various cross-correlation information and a crash-witness video. Like SAPFIX, Sapienz comments to developers in Phabricator, the backbone Facebook's Continuous Integration system[33], which is used for code review, handling more than 100,000 Diffs per week at Facebook [132]. Sapienz comments on commits (known as 'Diffs') to the central repositories for each app as they are submitted, using smoke builds from Facebook's 'Sandcastle' test infrastructure. More details on Sapienz can be found in the SSBSE 2018 keynote paper about its deployment at Facebook [6].

Although this generate and select process is effective and the debugging payload reported to developers for each crash leads to a high fix rate, the fixes take developer time and effort that can often be better spent on more creative programming activities that benefit from human insight, flexibility and domain knowledge. We observed that many of the fixes committed by developers to fix crashes reported by Sapienz are relatively small changes that could potentially be automated and so we looked

[33]http://phabricator.org

the research literature on automated fixing, with the aim of deploying SAPFIX; a fully end-to-end system to generate test cases that reveal faults and fix these faults automatically,

2.11.4 Infer

SAPFIX uses Infer to assist with localisation and static analysis of fixes proposed. Infer is deployed on the majority of Facebook code and based on Separation Logic and bi-abduction [235, 236], scaled to tens of millions of lines of code, thereby allowing Infer to find thousands of bugs per year [345]. Infer is also available as open source [57] and has been used elsewhere, including AWS, Mozilla, Spotify. Infer achieves its scalability through compositional program analysis; the analysis of a composite program is computed from the analysis of its parts [58]. Like Sapienz, Infer is deployed directly into Facebook's internal continuous integration system, where the two tools collaborate to highlight to developers those bugs on which they agree [6]. At the time of writing the Sapfix paper [213] such bugs have a 98% fix rate, largely we believe, because developers have a localisation of both the likely root causing fault (from Infer) and a consequent failure (from Sapienz). Nevertheless, even for such highly 'human fixable' bugs, engineering effort and skill could be better spent on other more creative and less tedious engineering activities, thereby motivating our interest in automated fault fixing through SAPFIX (Chapter 3).

2.11.5 FBLearner

Both Sapienz and SAPFIX are deployed on top of FBLearner, Facebook's Machine Learning (ML) platform through which most of Facebook's ML work is conducted [134]. Sapienz and SAPFIX use FBLearner Flow to deploy continuous execution. Facebook uses FBLearner for many other problems, including search queries, language translation (2,000 language pairs are translated serving approximately 4.5 billion translated post impressions every day), as well as speech and face recognition [134]. It is outside of the purpose of this thesis to explain FBLearner in more detail, but the infrastructure itself is covered in more detail elsewhere [134] and the use of FBLearner as a substrate on which to deploy search based testing is described in detail in the SSBSE 2018 keynote paper [6].

2.12 Automated Program Repair

SAPFIX is grounded in the approach to software engineering known as Search Based Software Engineering (SBSE) [126, 130]; the space of potential fixes to a software systems is considered to be a search space constructed from small modifications to an existing system under test. In the deployment of SAPFIX we do not claim any strong novelty in terms of the core repair algorithm. In fact, SAPFIX does not use any of the repair approaches from the literature on repair and SBSE, since their sophistication might have inhibited scalability. Instead we favoured using more simple approach in which a patch is simply a higher order mutant [146, 147] and we perform a single generation search over this space; essentially little more than random search, with some smart selection. As such, our results may best be thought of as a base line [131], against which to measure the advances we hope to see produced by future research and development.

SapFix is a simple realisation of Automated Program Repair, a topic that has been the subject of much research interest for over a decade [17, 185, 245], partly building on research on SBSE that has been a topic a research for more than two decades [126, 130]. With our SapFix approach we do not seek to make significant novel contributions to the underlying science of automated repair, but rather to demonstrate, explain and bear witness to the real world applicability these research agendas on Automated Repair, Automated Test Case Design and SBSE.

Much related work exists on the topic of repair alone, so we cannot hope to do justice to all of it here. Genetic improvement was successfully used for automated program repair, as we show in Section 2.3. We review the software transplantation based automated program repair approach in Section 2.13. In the remainder of this section, we briefly review other recent related work on Automated Program Repair and its differences and similarities to our deployment of SapFix. A more detailed survey can be found in the work of Monperrus [228]. According to the classification of Monperrus, SapFix is an offline behavioral repair approach.

Behavioral repair implies changing the source code of the buggy program. It requires an oracle to identify whether or not the bug is successfully fixed. Monperrus identifies three types of such oracle: test suites (the most closely related to SapFix), pre- and post- conditions, and abstract behavioral models [332].

Unlike the (many) other behavioral repair approaches in the existing literature [158, 185, 219, 245, 343], SapFix uses three different oracles to asses the quality and correctness of a fix: test cases from Facebook's CI, crash triggering sequences of UI events (similar to the work of Tan *et al.* [307]), and human reviewers. One novelty of our work derives directly from our industrial deployment; we are able to rely on expert engineers to act as the final arbiter of correctness in each case of a deployed repair.

This is the first time that professional engineers have played this role in the repair literature. It is also, simultaneously, an empirical evidence to support the claim that, at least there do exist automated program repairs that are, *ipso facto*, acceptable to expert professional software engineers.

Our SapFix approach targets Android NPEs and draws on templates automatically learned from human testers. Previously, Tan *et al.* [307] studied a set of Android crashes from Github to identify a set of 8 mutation operators that are often used by Android developers to fix bugs. One of their mutation operators is 'Missing Null Check', which is similar to our NPE mutation operators in the mutation fix strategy. Cornu *et al.*. have also targeted NPEs for repair [72]. NPEFix uses a predefined template-based approach similar with SapFix's templates.

SemFix [233] uses symbolic execution and code synthesis. Angelix [219] is an extension to Semfix that improves availability by optimizing the symbolic execution stage. PAR [158] also uses repair templates to fix common types of bugs in Java programs. One of their templates is also a "Null

Pointer Checker" that is parameterized by the variable name. PAR randomly applies the templates and validates the fix.

Nopol [343] is an automated bug fixing tool for two types of bugs: buggy `if` conditions and missing preconditions. In the case of missing preconditions, Nopol adds a guard (`if` statement), similar to our mutation fix operator for NPEs. Nopol synthesizes the fix, using oracles (input-output pairs) to guide component based patch synthesis [145].

Although SAPFIX represents the first industrial deployment of end-to-end repair (from automated test design through to fix deployment), there have been previous deployments of other forms of automated code change, both in industry, and to open source development communities. For example, Google's Tricorder system is reported to recommend fixes [272]. However, the experience report on the Tricorder ecosystem indicates that these fixes have the character of 'code smells' like an assignment operator used in a suspicious syntactic context. They are not detected and checked by automatically-constructed tests as with SAPFIX. Automated refactoring has also been widely studied [225] and has found deployment at scale in industry [341]. However, refactoring seeks to apply known-to-be semantically safe changes; essentially altering syntax without disrupting semantics. Therefore, although undoubtedly important, refactoring is less challenging than repair, which necessarily affects semantics as well as syntax.

The Repairnator system was first deployed in 2017 [317] to suggest repairs to Github Java projects. Repairnator uses existing test suites in these open source projects to identify crashes so, unlike SAPFIX, it does not offer end-to-end test generation to repair. However, like SAPFIX, Repairnator does provide for the continuous deployment of repair techniques at considerable scale.

2.13 Automated Software Transplantation

In this section we survey the existing software transplantation approaches and we formulate them in the terms of the transplantation formalism that we will introduce in Section 4.3. As a way to reuse code, manually transplanting code is a longstanding practice [118, 270]. Figure 4.9 captures both manual and automatic transplantation.

Table 2.1 reports the papers in the literature that propose different automated code transplantation approaches. Automated software transplants were used in the literature for different use cases. We identified two transplantation approaches for automated program repair, one transplantation approach for enhancing testing, one transplantation approach for security, and 12 transplantation approaches for functionality improvements . As Table 2.1 shows, the bulk of the transplantation work appeared after our ISSTA 2015 paper [30].

Automated Program Repair: there are a couple of automated transplantation approaches used for automated program repair [228]. In 2009, Weimer et al. [334] transplanted code from one location to a localised bug point within the same program. The goal was to find a form of automatic patching to fix bugs. However, in their work, the host and the donor systems were the same software systems.

Table 2.1: Automated software transplantation approaches in the literature.

Year	Venue	Paper	Use Case
2009	ICSE	Automatically finding patches using genetic programming [334]	repair
2012	JCV	In situ reuse of logically extracted functional components [226]	functionality
2014	EuroGP	Using genetic improvement & code transplants to specialise a C++ program to a problem class [248]	specialization
2014	SSBSE	Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system [127]	functionality
2015	ISSTA	Automated software transplantation [30]	functionality
2015	SSBSE	Automated transplantation of call graph and layout features into Kate [30]	functionality
2015	SSBSE	Grow and serve: Growing Django citation services using SBSE [149]	functionality
2015	PLDI	Automatic error elimination by multi-application code transfer	repair
2015	TR	Horizontal code transfer via program fracture and recombination [286]	functionality
2015	ECSA	Towards specifying pragmatic software reuse [212]	functionality
2016	FSE	Hunter: next-generation code reuse for Java [329]	functionality
2017	ISSTA	CPR: Cross platform binary code reuse via platform independent trace program [172]	functionality
2017	ICSE	Automated transplantation and differential testing for clones [172]	testing
2017	arXiv	Smartpaste: Learning to adapt source code [4]	functionality
2017	ACSA	Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps [346]	security
2017	FSE	CodeCarbonCopy [287]	functionality
2018	ICSE	Program splicing [201]	functionality

Thus, problems like namespacing conflicts, variable bindings, α–renaming, where not treated in their work. The amount of transplanted code was significantly lower in their case: one line code transplants. Weimer *et al.* do not use input (α_i) or output (α_o) adapters.

More closely related to our work, Sidiroglou-Douskos [288] used code transplants for automatic bug fixing, implementing a tool called `CodePhage`. In their case, the transplants contain only one conditional statement (called *check* in their work).

Having a *recipient* program that contains a bug, and a *donor* program that do not contain this bug, they assume that the donor program checks, at some point during its execution, an *if* condition that eliminates that bug. Once identified that condition, they identify an insertion point in the recipient system, where all the variables involved in the `if`'s predicate are available. They automatically translate the check from the namespace of the donor system, into the one of the host system, and add that check at the identified insertion point. If all these are successfully identified, they enclose the check into an `if` statement, and add an `exit(1);` instruction on the true branch of the `if`.

For CodePhage's approach to work, the donor and the host system must be closely related: both program must accept the same inputs, such that both of them are able to process a seed input (that is processed correctly by both the recipient and the donor), and a bug–triggering input (that triggers the bug in the recipient system, but is correctly processed by the donor system). More then this, the bits of the input bytes relevant for the identified check, must appear in the same order in both the recipient, and the donor system, while in the recipient must exists a program point, where all the values involved in the check condition are available. To construct α_i, CodePhage extracts the computations c_d from a vein and H's computations c_h from entry to $\square$, the implantation point. They use an SMT solver to find the largest match between c_d and c_h, then remove that match from c_d and construct α_i from the unmatched computations in c_d. The transplant fails if no match is found.

As opposite to their work, we do not have these limitations: we do not assume any relation between the donor, and the host system, as well as we do not assume any constraints about the individual inputs of the host, and donor system; or about the relation between the inputs of the host and the donor system. While in their case only one line of code from the donor system is transplanted (with possible some additional binary computations, on the variables involved in the check, but still contained in the same `if` statement), our work transplants massive amount of code (see Section 4.7, Section 4.9, and Section 4.11.2). We transplant genuine new non-trivial functionality into the host system, while CodePhage transplants only one predicate of an `if` statement, for exiting the execution of the recipient program on the true branch. While μSCALPEL and τSCALPEL both work on source code, CodePhage works on binary code.

Security: code transplants were also used for security purposes, to assess the malware detection capabilities of existing malware detectors. Yand *et al.* [346] use code transplants to test malware detection systems. They use software transplantation to automatically mutate malware at the bytecode level to generate new malware variants that preserve the malicious behaviour. Their approach evaluate whether or not the malware detectors can detect the malware variants that transplantation produced. They evaluate their approach on 1,935 Android bening apps and 1,917 malware. Their results show that software transplantation is effective at avoiding malware detection: out of 696 malware generates that transplantation produced, VirusTotal [313] fails to detect 565, AppContext [347] fails to detect 153, and Drebin [20] fails to detect 518.

Yand *et al.*'s transplantation framework use backward program slicing to construct the input adapter α_i for the organ to transplant, the malicious behavior in their case. Programs slicing scales in their case, as the malware code that they transplant do not usually have dependencies on the rest of the donor system.

Testing: code transplants were also used to allow test cases reusal on a transplanted organ. Zhang *et al.* [352] automatically transplant a clone into the place of its tested counterpart to reuse test cases. They construct α_i and α_o using five heuristics that propagate the data from H's data structures into O's data structure (α_i) and vice versa (α_o). For example, when a free variable in O has the same type (or castable) as a variable in H available at I_P, they assign the value of the variable in H to it.

Functionality Transplantation: is the main use case of code transplants in the software engineering literature. Table 2.1 reports 12 different approaches that implement code transplants for functionality reusal. Our monolingual (Chapter 4) and multilingual (Chapter 5) are approaches for functionality transplantation.

Miles *et al.* [226] proposed an approach for extracting logical components, from a running application, that are implementing a feature of interests. Using manual annotations, the authors are using a debugger for starting in-situ a specified component of a software system (without actually extracting it from the *donor* system). The annotations might specify the program state from where the in-situ logical component must start, the entry point of the component, and its exit point. Having these provided, their approach would execute the logical component in isolation, but from the binary of the entire *donor* program. Miles *et al.*'s approach do not use adapters, as they do not implant the organ in a different program. Their transplantation function $\mathscr{T}$ is identity in their case.

More recently, in 2014, the automatic transplantation of code across different versions of the same program was demonstrated: the donors were versions of MiniSAT grafted into a specific MiniSAT version (the 'host') [248, 249]. The resulting code outperformed all other versions for our SAT application problem, even though its competitors had all been extensively human-optimised over many years. This work won the silver medal at the GECCO 2014 Humies. In this work, Petke *et al.* do not use adapters, as the approach transplants code from a different version of the *host*, but from the same context: $P_D = P_H \wedge Q_D = Q_H \wedge \alpha_i = \varepsilon \wedge \alpha_o = \varepsilon$.

We are the first to transplant functionality between *different* systems. We believe we are the first to do this, although previous work [127] has grafted new functionality (grown from scratch) into existing systems. There, GI was used for evolving small functionality from scratch, starting from some hints provided by the user. The hints provided by the user, included: functions to be used in the new functionality, order of the function calls, and input received / output of the suggested function calls.

Recently, Sidiroglou-Douskos *et al.* [286] proposed a horizontal code transfer technique, analogue with the biological horizontal gene transfer [11, 29, 154, 156]. Their approach is a horizontal code transfer technique, called *program fracture and recombination*. It can be used for automatically locating and transferring computations between multiple applications. In their view, a program is *fractured* into multiple *shards*. A *shard* is a piece of program that implements some specific computations, or a functionality. Example of shards could be functions, or modules into a system system.

Having fractured the program, and obtained its shards, there approach might be used under three use cases: 1) shard insertion: insert a shard from a *donor* program into a *recipient* one; 2) shard deletion: delete an existing shard in a program; 3) replace a shard from another one in the same program, or in another program.

Autotransplantation is most similar with their first use case. However, in their approach, they do not compute the *vein*, as in autotransplantation. That's it, for them, the shard is directly inserted at the insertion point, into the host system, without any modifications. Thus, they do not use input or output adapters and the transplantation function simply moves the code, without any adaption: $P_D = P_H \wedge Q_D = Q_H \wedge \alpha_i = \varepsilon \wedge \alpha_o = \varepsilon$. For this approach to be feasible in practice, the host system must provide a very similar execution environment as the donor system, while the actuals (parameters used for calling the shards in the host) must be identical with the formals[34] in the original implementation of the shard in the donor system. Autotransplantation does not have any of these limitations, by not making any assumptions about the host, donor, and the relation between them. Their approach is similar with copy-paste of a section in the donor system. Currently, their prototype was evaluated only on a trivial, artificial program.

Maras *et al.* [212] propose a general approach for software reuse, similar with the steps in our autotransplantation approach, but without implementing it. Their framework contain three big steps: feature localization to identify the code to transplant (*i.e.* organ); code analysis and modification, similar with our organ adaptation via GP approach; and finally feature integration that is similar with the implantation step in our autotransplantation approach.

In 2016, Wang *et al.* [329, 330] proposed Hunter, a framework for code transplants. Hunter takes as inputs the desired type signature of the organ and a natural language description of its functionality. First, Hunter uses any existing code search engine (Pliny [81] in the current implementation) to search for a method to transplant in a database of software repositories using the type signature and the natural language description. Next, Hunter computes a type similarity metric between the found methods and the desired type signature. For each of the candidate, Hunter matches the type signatures of the candidate with the desired type signature by using the type distances: it matches variables in the desired type signature to the one with the smallest type distance in a candidate function. Further, Hunter generates adapter functions to transforms the types from the desired type signature into the type signatures of the candidate functions. These adapters are the input adapters α_i. To construct them, Hunter uses off the shelf code synthesisers. The current Hunter implementation uses SyPet [100]. Hunter does not use output adapters. The transplantation function moves the code without adaption to the host context.

Kwon et al. propose CPR [172] to transplant a program on different platforms. CPR realizes software transplantation by synthesizing a platform independent program from a platform dependent program. The resulted program can run on different platforms. To synthesis the platform independent program, CPR uses PIEtrace [173] to construct a set of trace programs. PIEtrace constructs a trace program from a regular program by monitoring a concrete execution of a program. PIEtrace captures the control flow path and the data dependencies observed during a concrete execution. Further, PIEtrace replaces all the platform dependencies with the concrete values that it observed during the concrete execution. A trace program is a *closed* program: a program that does not take any inputs. The PIEtrace produced trace program replicate the concrete execution on which it was produced. To produce an *open* program that can process any inputs, CPR calls PIEtrace multiple times to produce a set of trace programs. Further, CPR merges all these trace programs together to handle any input, by replacing the concrete values observed during the executions, with input variables. CPR does not use

[34] https://msdn.microsoft.com/en-us/library/f81cdka5.aspx

input / output adapters, as it does not transplant an organ in a host: CPR transplants an entire program on different platforms.

Allamanis *et al.*'s SMARTPASTE [4] automates the variable matching subtask of transplantation. SMARTPASTE takes, as input, a piece of code (O) to paste it into $\square$. O does not have a context and they do not use adaptors. They replace variable names with holes and use a deep neural network to fill them. Allamanis*et al.* [5] use Gated Graph Neural Networks [189] to predict the correct variable name in an expression and to identify when a variable name is missuses. Their approach can help automated software transplantation to connect the variable names in the host to the ones in the organ by identifying which host variables an organ should use.

CodeCarbonCopy [287] applies CodePhage [288] to the task of transplanting functionality and extends it to construct α_i to handle additional language constructs such as arrays. We discussed CodePhage in the automated program repair paragraph.

More recently, Lu *et al.* [200, 201] introduced program splicing. Program splicing is a framework to automate the process of code transplant: copy, paste, and modify the organ. In their approach, the implantation point in the host is a draft that contains *holes*, natural language comments, and an underapproximation of the organ semantics such as test cases or API call sequence constraints. Program splicing looks into a database of programs to identify a small number of programs snippets (*i.e.* statements and expressions) that are relevant to the current transplant task. Lu *et al.* use a k-nearest neighboring algorithm [246] to identify related program snippets for the program draft and to rank them. Finally, program splicing enumerates over the relevant program snippets, to fill the holes in the draft.

To our knowledge, our automated software transplantation approach is the first time that code and its functionality has been automatically transplanted between unrelated non-trivial, real-world systems, both across the same programming language (μTrans) and across different programming languages (τTrans). Our work also departs from previous work on genetic improvement and other attempts to improve the non functional properties of systems [17, 128, 178, 237, 303, 338] because we seek to augment the functionality offered by the host, rather than improving its performance.

2.14 Conclusion

This chapter reviewed the relevant automated program repair literature, with a focus on automated software transplantation that is the kind of automated program repair on which this thesis focuses. Our approach for automated software transplantation builds on the existing literature in search based software engineering, genetic improvement, evolutionary computation, program slicing, code synthesis, and code translation.

Our automated program repair work in SAPFIX (Chapter 3) would benefit from new, scalable approaches in the automated program repair literature (Section 2.12). We could integrate and evaluate such approaches into our scalable automated program repair framework that works on system on millions of lines of codes, used by billions of people around the world, would the approaches be able to scale at such code base and produce candidate fixes in a reasonable amount of time and with reasonable resource requirements.

Automated software transplantation is a form of genetic improvement (Section 2.3) and a form of automated program repair (Section 2.12) that repairs the functionality of a program that was initially missing a required functionality. Genetic improvement was previously successfully used to improve the non-functional properties of a software system, such as execution time [178]. There is some work on using genetic improvement to *transplant* a program in a different language and on a different platform [176] when the source and target languages are very similar (*e.g.* C to CUDA). Genetic improvement was successfully used for automated bug fixing [185], but Qi *et al.* [257] discovered a set of flaws in the evaluation of automatic bug fixing tools. Genetic improvement was also used for software transplantation, but for small functionalities [127], or in the contexts where the host and donor programs were the same and the code was transplanted into a very similar context with its source context [248]. Our automated software transplantation approach transplants functionalities when the host and donor program significantly differ and are potentially written in very different programming languages.

Our monolingual and multilingual automated software transplantation approaches would benefit from advances in areas such as evolutionary computation (Section 2.1) or program slicing (Section 2.4) that it uses in the transplantation process. Our multilingual software transplantation approach employs code translation (Section 2.6) and code synthesis (Section 2.5). Autotransplantation assumes the existence of a donor program to implement the desired, missing functionality. Recent work in code search (Section 2.7) would remove the burden from the user of autotransplantation of finding such a suitable donor. Once a transplant is done, a concern that raises is about allowing a transplanted organ to benefit from developers' improvements over the organ in the donor. Here autotransplantation might benefit from the relevant work in maintaining a transplant in-sync with its implementation in donor area (Section 2.8).

The most of the existing work tends to improve non-functional characteristics of a software system. Since our first software transplantation paper [30] was published in 2015, the research community published work in software transplantation that we reviewed in Section 2.13. Code transplants were tried prior to our ISSTA paper [30], independent by us, but in those cases the transplanted code was small in size, and often trivial [127, 286, 288]; or the source and the target contexts for a transplant were very similar [248]. The majority of autotransplantation papers appeared after our ISSTA paper [30] (Table 2.1)

The most related software transplantation approach prior to our work is the work of Sidiroglou-Douskos *et al.* [286, 288]. While in their work, Sidiroglou-Douskos *et al.* transplanted very small amount of code (only one `if`'s predicate), our approach scaled at transplanting over 23k lines of code in one of our case study (see Section 4.9).

Our literature review shows that the current software engineering literature lacks contributions in the process of automated software transplantation of real world, non-trivial functionalities, across diverse hosts and donor programs, potentially written in diverse programming languages. The reviewed related work tackles the transplantation problem in a limited context, but however, the overall process of autotransplantation is still manual, tedious, and error prone. Our work in autotransplantation is both novel, and useful (as our experiments show). The software engineering literature might benefit a lot from these work, while autotransplantation might simplify a lot the work of software engineering practitioners.

Chapter 3

SapFix: Automated End-to-End Repair at Scale

In this chapter we present our approach for automated program repair that works at the scale of systems of millions of lines of code that are used by billions of people around the word. We implemented our approach in a tool called SAPFIX capable of automatically producing fixes, validating them, and landing them into production. The human software engineer acts as the final gatekeeper deciding whether or not to accept the fixes that SAPFIX produced. This chapter is based on our ICSE SEIP 2019 paper [213].

This chapter is organized as follows: Section 3.1 introduces the problem of automated program repair; Section 3.2 describes the SAPFIX system itself; Section 3.3 reports the results of using SAPFIX to automatically fix, validate, and land into production fixes across 6 key Android Facebook applications; and Section 3.4 concludes this chapter and briefly describes some future work ideas.

3.1 Introduction

Automated program repair seeks to find small changes to software systems that patch known bugs [184, 247]. Automated program repair approaches use test cases to guide and validate their fixes for such bugs. Sapienz [210] is a search based software testing tool that has been deployed at scale [6, 132] and finds hundreds of crashes per month for which it generates test cases to reveal them. In these chapter we used these test cases that Sapienz automatically designs to guide our automatic program repair approach.

We have started to deploy automated repair, in a tool called SAPFIX, to tackle some of the crashes that Sapienz finds. SAPFIX is the first fully automated deployment of testing and repair at scale in continuous integration and deployment in industry. SAPFIX automates the entire repair life cycle end-to-end with the help of Sapienz: from designing the test cases that detect the crash, through to fixing and re-testing, the process is fully automated and deployed into Facebook's continuous integration and deployment system.

The Sapienz deployment at Facebook, on which SapFix rests, currently tests the Facebook app itself (for Android and, since September 2018, also for iOS) and the Android Messenger, Instagram, FBLite, Workplace and Workchat apps. This chapter focuses on the deployment of SAPFIX on six key Android apps, for which the Sapienz test input generation infrastructure is now reasonably mature, relative to the more recent deployment on iOS. These six Android apps consist of tens of millions of lines of code and are used by hundreds of millions of monthly active users worldwide to support communication, social media and community building activities.

They constitute some of the largest, most complex and most widely used apps in the app store. Indeed, they are some of the most widely used software systems in the history of Software Engineering.

The human software engineer acts as the final gatekeeper in the SAPFIX repair process, deciding whether or not to accept the proposed fix into production, as one would expect. This ensures proper human oversight of an otherwise, entirely automated process. In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques, nevertheless the SAPFIX core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [147, 148], search-based software testing [6, 125, 218], and fault localisation [244, 350] as well as existing developer-designed test cases. We also needed to deploy many practical engineering techniques and develop new engineering solutions in order to ensure scalability.

SAPFIX combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a deletion-as-last resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer's static analysis and the localisation infrastructure built specifically for Sapienz [6]. SAPFIX is deployed on top of the Facebook FBLearner Machine Learning infrastructure [134] into the Phabricator code review system, which supports the interactions with developers.

We report our experience, focusing on the techniques required to deploy repair at scale into continuous integration and deployment. We also report on developers' reactions and the socio-technical issues raised by automated program repair. We believe that this experience may inform and guide future research in automated repair. We conclude with a set of open problems and challenges for the research community.

Our ultimate goal is the vision of addressing the FiFiVerify challenge [125, 132]; Finding faults, Fixing them, and Verifying the fixes entirely automatically. The ICST 2015 keynote paper [125] is but one formulation of this scientific challenge; there are other formulations of closely related challenges, such as the DARPA Cyber Grand Challenge (CGC)[1], which specifically targets security fixes.

The SAPFIX project is a small, but nevertheless distinct advance, along the path to the realisation of this ultimate vision. The primary contributions of this work are:

[1]https://www.darpa.mil/program/cyber-grand-challenge

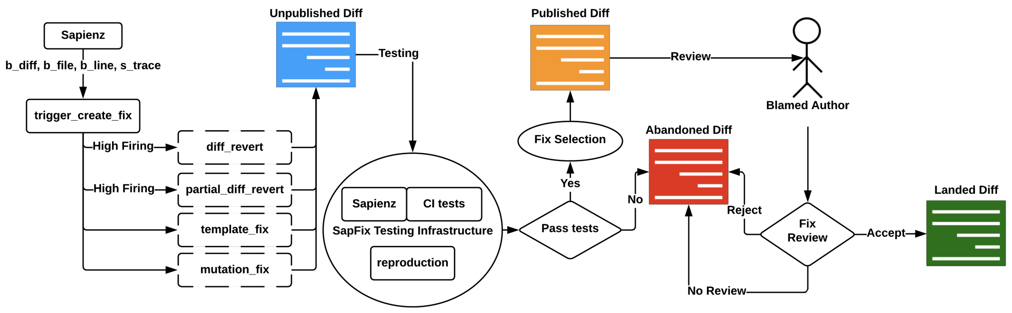


Figure 3.1: SAPFIX workflow. When Sapienz triages a crash to SAPFIX, *trigger_create_fix* triggers the four fix strategies that create fix candidate (unpublished) *diffs*. Next, SAPFIX tests the candidate fixes. The fix selection stage uses heuristics to select one diff out of the ones that pass all the tests and further publishes it to notify the most relevant software engineer and add him or her as reviewer If the reviewer accepts the fix, SAPFIX lands it into the production workflow of the Phabricator Continuous Integration system. SAPFIX abandons the fix candidate if the developer rejects it or he or she fails to review it within 7 days.

1. The first end-to-end deployment of industrial repair;
2. The first combination of automated repair with static and dynamic analysis for crash identification, localisation and re-testing;
3. Results from repair applied to 6 multi-million line systems;
4. Results and insights from professional developers' feedback on proposed repairs.

3.2 The SAPFIX System

This section describes the SAPFIX system itself, its algorithms for repair and how it combines the components outlined in the previous section.

3.2.1 The SAPFIX Algorithmic Workflow

Figure 3.1 shows the main workflow of SAPFIX. Algorithm 2 is the main algorithm of SAPFIX that drives the automated bug fixing process. Using Phabricator, Facebook's continuous integration system, developers submit changes (called 'Diffs') to be reviewed. Sapienz, Facebook's continuous search based testing system, selects test cases to execute on each Diff submitted for review [6]. When Sapienz triages a crash to a given Diff, SAPFIX executes Algorithm 2. Line 1 in Algorithm 2 establishes the priority of fixes according to the strategy that SAPFIX used to produce them. When multiple fix strategies produce patches that pass all SAPFIX's tests, we select fixes only from the top priority strategy to report to developers. This prioritisation approach avoids polluting developers' review queues.

The *template_fix* and *mutation_fix* strategies (mentioned at Line 1 of Algorithm 2) choose between template and mutation-based fixes, favouring template-based fixes, where all else is equal, but taking account of results from Infer static analysis and also from linter reports on candidate

Algorithm 2 *trigger_create_fix*, SAPFIX's trigger fix creation algorithm.

```
Input   b_rev, the buggy revision
        b_file, the blamed file: the file that contains the crash location
        b_line, the blamed line: the line of the crash
        s_trace, the stack trace of the crash
        mid, the mid of the crash we are trying to fix
        b_author, the blamed author: the author of b_rev
        ?buggy_expressions, buggy expressions that Infer gives us. When we do not have Infer data, this argument is null
        high_firing_t, the high firing threshold
Ensure: P, a list of revisions that fix the crash under SAPFIX's testing
 1: strategy_priority := [template_fix, mutation_fix, diff_revert, partial_diff_revert]
 2: $C_P := \emptyset$ {$C_P$ is the list of candidate fixes}
 3: if is_high_firing(mid, high_firing_t) then
 4:     $C_P$ += diff_revert(b_rev)
 5:     $C_P$ += partial_diff_revert(b_rev, b_file, b_line)
 6: $C_P$ += template_fix(b_rev, b_file, b_line, buggy_expressions)
 7: $C_P$ += mutation_fix(b_rev, b_file, b_line, buggy_expressions, s_trace)
 8: $P := \emptyset$ {$P$ is the list of patches that fix the bug}
 9: for all $p \in C_P$ do
10:     if $\neg$repro_crash(p, mid) $\wedge$ pass_sapienz(p) $\wedge$ $\neg$sapienz_repro_mid(p, mid) $\wedge$ pass_ci_tests(p) then
11:         $P$ += $p$
12: for all s $\in$ strategy_priority do
13:     $P_s$ := filter(P, strategy($p \in P$) = s)
14:     if $P_s \neq \emptyset$ then
15:         $p_s$ := select_patch($P_s$)
16:         publish_and_notify($p_s$, b_author)
17:         for all $p_u \in P_s \setminus \{p_s\}$ do
18:             publish_and_comment($p_u$, $p_s$)
19:         return $P_s$
20: return null
```

fixes. Template fixes come from another tool that generates patches similar to the ones that human developers produced in the past; the details will be described in a subsequent publication. For the purposes of understanding the SAPFIX deployment, it can be assumed that SAPFIX has available to it, a set of template fix patterns harvested from previous successful fixes deployed by developers.

If neither template-based nor mutation-based approach produces a patch that passes all tests, SAPFIX will attempt to revert Diffs that result in high-firing crashes. Lines 3–5 in Algorithm 2 trigger the Diff revert strategies. SAPFIX triggers these strategies only for high-firing crashes that block Sapienz and other testing technologies and therefore need to be deleted from the master build we are testing as soon as possible (even if they would never ultimately leave the master build and make it to production deployment). If the Diff added lines of code then reversion is simply an attempt at side-effect free deletion. If the Diff removed lines of code, then reversion would add them back. Either way (and for everything in-between), conceptually speaking, reversion means to 'delete the Diff'.

Between the two available Diff reversion strategies, SAPFIX prefers (full) *diff_revert*, because *partial_diff_revert* is expected to have a higher probability of knock-on adverse effects due to dependencies between the changes in the Diff that introduced the crash. However, (full) *diff_revert* might fail because of merge conflicts with the master revision (new Diffs land every few seconds, while fix reporting can take up to 90 minutes (see Figure 3.3). In those cases we use *partial_diff_revert*. The changes that *partial_diff_revert* produces are smaller and thus less prone to merge conflicts.

The recognition of a crash and the distinction between different crashes requires a 'crash hash'; a function that groups together different crashes according to their likely cause. This is a non-trivial problem in its own right. Facebook uses a crash hash called a 'mid', the technical details of which are described elsewhere [6]. For this discussion, the important characteristic of a 'mid' is that it is an approximate (but reasonably accurate) way of identifying unique crashes. It can be thought of, loosely speaking, as a crash id. Improving the accuracy of such crash hashes remains an interesting and important challenge for future research [6].

Of course, we favour a fix rather than to simply attempt to delete the offending code, but when the other fix strategies fail to fix a high firing crash, SAPFIX suggests a Diff revert fix. $is\_high\_firing(mid, threshold)$ identifies high firing crashes: it returns true if the crash with id mid fires more than $threshold$ times, and false otherwise. Finding a way to delete the right code without affecting other subsequent Diffs is also a non-trivial problem in a large scale and rapidly changing code base, where new Diffs land every few seconds. This problem may also benefit from further attention from the research community.

Lines 6–7 in Algorithm 2 trigger the template and mutation fix strategies. Lines 9–11 looks at the candidate fix patches in C_P for the ones that indeed fixed the crash, without introducing new bugs.

To identify whether a patch fixes a crash, SAPFIX uses $repro\_crash$ and $sapienz\_repro\_mid$. $repro\_crash(rev, mid)$ tries to reproduce mid in the revision rev using Sapienz's reproduction workflow. We cannot always assume that we have available tests that reproduce a given crash, due to the well-known problem of test flakiness [132, 202]. Therefore, SAPFIX also uses $sapienz\_repro\_mid(rev, mid)$ to inspect the results of regular Sapienz runs over rev to see if any of them found mid. Infer is also re-executed (automatically) on the patches it has detected as a sanity check that static analysis also no longer identifies the issue that SAPFIX seeks to fix.

To identify whether a patch might also introduce new crashes or other issues, SAPFIX runs Sapienz multiple times over the candidate fix in $pass\_sapienz(rev)$. Finally, $pass\_ci\_tests(rev)$ inspects the results of (previously existing) unit, integration, and end-to-end tests in the Facebook continuous integration and deployment infrastructure. If all these tests pass, SAPFIX considers the patch to be a successful candidate to report to engineers and adds it to P, at Line 11 in Algorithm 2.

Lines 12–19 in Algorithm 2 publish the successful candidate patches. SAPFIX selects one of the published candidates and requests a reviewer for this candidate through the Phabricator code review system. The reviewer is chosen to be the software engineer who submitted the Diff that SAPFIX attempted to fix. This is the engineer who most likely has the technical context to evaluate the patch.

Other relevant engineers are also subscribed to each Diff published by SAPFIX to oversee the review process, according to heuristics implemented, as standard for all Diffs, in the Facebook code review process. Furthermore, some developers specifically ask to be subscribed to (some or all) fixes, by opting in with a so-called 'butterfly' subscription rule. As a result, all Diffs proposed by SAPFIX are guaranteed to have at least one (suitably qualified) human reviewer, but may have many more, through these other routes to Diff subscription.

The function $strategy(p)$ returns the strategy that SAPFIX used to produce p. Line 13 selects in P_S, all the successful patches that the top priority strategy produced. Next, at line 15, $select\_patch(P_s)$

Algorithm 3 *diff_revert*, SAPFIX diff revert algorithm.

```
Input    b_rev, the buggy revision that Sapienz blamed
Ensure:  p, a revision that reverts b_rev
1: return  $vcs\_backout(b\_rev)$
```

Algorithm 4 *partial_diff_revert*, SAPFIX partial diff revert algorithm.

```
Input    b_rev, the buggy revision that Sapienz blamed
         b_file, the blamed file: the file that contains the crash location
         b_line, the blamed line: the line of the crash
Ensure:  P, the list of partial revert revisions for b_rev
1: $P := \emptyset$
2: $\text{complete\_revert} = vcs\_diff(\text{b\_rev}, vcs\_parent(\text{b\_rev}))$
3: $P \mathrel{+}= create\_rev(extract\_file\_change(\text{complete\_revert}, \text{b\_file}))$
4: $P \mathrel{+}= create\_rev(extract\_hunk\_change(\text{complete\_revert}, \text{b\_file}, \text{b\_line}))$
5: return  $P$
```

selects in p_s the top priority patch from P_s. Currently, $select\_patch(P_s)$ uses the following heuristics: select the fix that Sapienz executed the most often; select the fix for which Sapienz executed the buggy statement the most often; select the smallest fix. On Line 16, SAPFIX publishes p_s and notifies the developer by calling $publish\_and\_notify(p, b\_author)$.

Finally, on Lines 17–18, Algorithm 2 publishes the rest of the patches from P_s and comments with a preview of them on p_s. $publish\_and\_comment(p, p_s)$ publishes the candidate fix p and adds an inline preview of p on the selected candidate fix p_s.

Algorithm 3 is the diff revert algorithm that SAPFIX uses to completely revert the blamed diff. Algorithm 3 calls the version control system's backout functionality in $vcs\_backout(rev)$, to create a revision that reverts b_rev. In the case of a merge conflict with master, *diff_revert* does not produce a patch.

Algorithm 4 is the algorithm that SAPFIX uses to partially revert a blamed diff. Algorithm 4 produces two patches: the first patch reverts the file containing the buggy statement and the second one reverts only the hunk that contains the buggy statement. On line 2 Algorithm 4 uses the version control system's "diff" functionality to produce the patch that reverts the entire b_rev. $vcs\_parent(\text{rev})$ returns the parent revision of rev. On line 3, SAPFIX extracts the changes that affect b_file from $complete\_revert$. The method *create_rev* creates a revision in our CI. On line 4, SAPFIX calls the method *extract_hunk_change* to extract the hunk that affects b_line.

Algorithm 5 is SAPFIX template fix algorithm. Algorithm 5 calls getafix at line 1 to produce candidate bug fixing revisions.

Algorithm 6 is the SAPFIX mutation-based fixing algorithm. Algorithm 6 currently only supports fixing NPE crashes. We are currently in the process of extending the mutation strategies to cater for other crash categories, but we have already witnessed considerable success with NPE-specific patching alone, which is encouraging. On Line 1, Algorithm 6 calls *extract_crash_category* to identify

Algorithm 5 *template_fix*, SapFix template fix algorithm.

```
Input    b_rev, the buggy revision that Sapienz blamed
         b_file, the blamed file: the file that contains the crash location
         b_line, the blamed line: the line of the crash
Ensure: P, the list of bug fixing revisions
 1: return getafix(b_rev, b_file, b_line)
```

Algorithm 6 *mutation_fix*, SapFix mutation fix algorithm.

```
Input    b_rev, the buggy revision that Sapienz blamed
         b_file, the blamed file: the file that contains the crash location
         b_line, the blamed line: the line of the crash
         s_trace the stack trace of the crash
         ?buggy_expressions candidate buggy expressions that Infer gives us. When we do not have Infer data, this argument is null.
Ensure: P, the list of bug fixing revisions
 1: crash_category := extract_crash_category(s_trace)
 2: if crash_category $\neq$ "NPE" then
 3:     return $\emptyset$
 4: if buggy_expressions $\neq \emptyset$ then
 5:     return create_rev(add_null_check(b_file, b_line, buggy_expressions))
 6: else
 7:     $C_{buggy}$ := top_of_s_trace(s_trace, b_file, b_line) ?
        extract_dereferences(b_file, b_line) : extract_args(b_file, b_line)
 8:     P := $\emptyset$
 9:     for all $c \in C_{buggy}$ do
10:         P += create_rev(add_null_check(b_file, b_line, c))
11:     return P
```

the category of crash. If the category is not "NPE", Algorithm 6 returns the empty set. To extract the crash category, *extract_crash_category* looks at the short message on the stack trace.

Lines 4–5 in Algorithm 6 check whether a more precise cause of the NPE is known: *i.e.* which expressions in the buggy statement caused the NPE by taking the value null. This information can be obtained from Infer, in cases where both Infer and Sapienz find the same NPE. When that happens SapFix creates a single patch that guards *b_line* with null checks for the buggy expression. On Line 5, Algorithm 6 creates a revision for this patch. The method *add_null_check* uses eclipse JDT to parse the AST of the buggy file and to add the null check before the buggy statement.

Lines 7–11 in Algorithm 6 handle the case when we do not know which expressions are buggy. In this case, SapFix identifies all the expressions in the buggy statement that can potentially cause an NPE. For each such expression, Algorithm 6 produces a candidate patch. SapFix tries each of two simple mutations, which either return null or protect potentially null-valued expressions with a null check. The surrounding Facebook testing infrastructure will subsequently tend to reject those patches that do not actually fix the bug (such as those patches that inadvertently attempt to 'fix' the wrong expression). Therefore, failure to fully localise the buggy expression tends to affect efficiency but not effectiveness.

Infer helps to localise the likely NPE-raising expression, but dynamic analysis can also help here, where Infer signal is unavailable. Specifically, we analyse the position of the blamed line of code in the stack trace (*b_line*) at Line 7 to obtain C_{buggy}, the set of candidate buggy expressions. If *b_line* is at the top of the stack trace then *top_of_s_trace*(*s_trace*, *b_file*, *b_line*) returns true. In this case, SapFix need only attempt to fix expressions that are de-referenced, because the program execution does not continue after *b_line*. *extract_dereferences* extracts only the expression de-referenced at *b_line*.

If *b_line* is not at the top of the stack trace, it means that one of the arguments of a function called at *b_line* is presumed to have caused the NPE (further up in the stack trace). In this case SAPFIX need only attempt to fix the arguments of functions in *b_line*. *extract_args* extracts the function arguments at *b_line*.

Finally, Lines 9–10 in Algorithm 6 produce a patch, for each candidate buggy expression in C_{buggy}. The patch guards *b_line* with null check for the candidate buggy expression.

The most common fixes that the mutation-fix and template-fix algorithms produced are very simple fixes such as early returns:

```
if (p == null){
  return value;
}
```

or adding a null check that guards dereference of potentially null values:

```
if (p != null){
  p.method();
}
```

A concern that such fixes raise is about the fact that they might mask the symptoms of a bug that manifest as a crash at the line of code where the fix happened, rather than fixing the root cause of the crash. Indeed, potentially in a lot of the cases the root cause of the problem was an unexpected value of `null` for `p`, rather than the dereference itself. In some cases such fixes that SAPFIX produces could replace the crash by a loss of functionality, possibly leaving the application in a state where it would have to restart anyway. For example, consider the case of an NPE that happens early in the code, before *News Feed* load. Early return here would just display the user an empty screen which is probably not very useful. In the Facebook ecosystem this is less of a concern as before suggesting a fix, SAPFIX runs all the test cases that are relevant for the changed code. Big lost of functionality that would affect the experience of our users would make some of our test cases to fail in general as the quality of our test suites is pretty high.

In a large, real world, complex application as is the case for the Facebook applications on which we executed SAPFIX, there are many crashes happening every day and not enough bandwidth of our developers . We believe is better to apply such fixes that SAPFIX does, in cases where the functionality of the application is not hugely affected by early returns for example, rather than letting the application crash and highly affecting the experience of our users. Once the developers got free time they can attempt a more *root cause* fix than SAPFIX does. We believe SAPFIX fixes are reasonable for two reasons:

1. For all our fixes that landed in production we used human code reviewers as the last gate keeper. Thus, a professional software engineer looked at the fix and deemed it as a good fix.

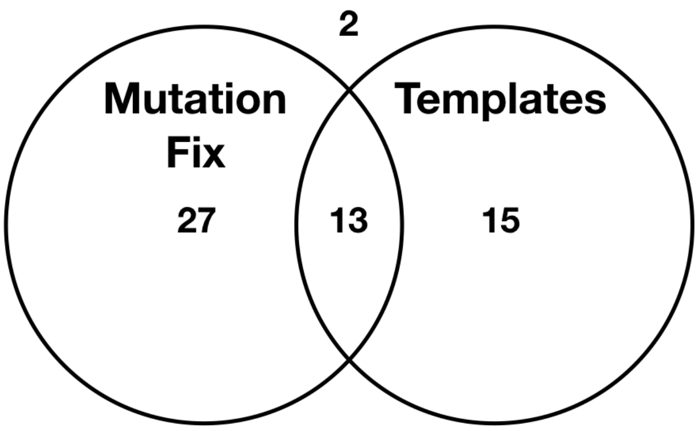


Figure 3.2: The number of times mutation-based fix strategy and templates produced at least one patch to pass all tests.

2. The template fix strategy looks at past developer fixes in our source control history. Thus, also human developers fix null pointer exceptions mostly with such simple fixes as SAPFIX did.

3.3 Results

Table 3.1 presents the results of applying SAPFIX over a period of three months to tackle NPEs detected by Sapienz as they were submitted for code review. Each row denotes a crash tackled by SAPFIX. Naturally, we periodically update the pool of template fixes (something that occurred once during the first three months of deployment, on the 9th. of August 2018 as shown in Table 3.1). In total, to tackle the 57 crashes reported to SAPFIX, 165 patches were constructed, of which roughly half were constructed using templates and half using mutation-based repair. Of these 165 patches, 131 correctly built and passed all tests and were thus fix candidates. Of these 131 candidates, 55 were reported to developers, covering 55 of the 57 crashes tackled by SAPFIX.

Figure 3.2 reports how many times the template-based and mutation-based strategies produced at least one candidate fix for each of the 57 different crashes from Table 3.1. Although SAPFIX favours templates overall, it triggers both strategies to be able to re-test all patches produced in parallel. Triggering both the fix strategies also allows us to evaluate these two strategies in isolation. Our results suggest that having both in the pipeline leads to better overall success: In 55 of the 57 fix attempts, either the mutation-based fix strategy *or* the templates produced at least one fix candidate. Only in two cases did *both* strategies fail (one failed to build and one failed re-testing). In 13 cases, both the fix strategies produced at least one fix candidate. In isolation, the mutation-based fix strategy produced at least one fix candidate for 40 cases, while the templates did so in 28 cases. In 27 cases the mutation-based fix strategy alone was able to produce a fix candidate, while in 15 cases the templates alone produced a fix candidate. The template-based fix strategy usually failed to produce a candidate fix because of lacking automatically mined based on human fixes templates for particular classes of fixes, such as guarding for null a function argument that when *null* causes a crash further on the call stack.

Initial reactions were strongly positive: On seeing the very first SAPFIX-proposed patch, the developer reviewing the patch commented: '*Definitely felt like a living in the future moment when it sent me the diff to review. Super cool!*'. As can be seen from Table 3.1, about half the fixes proposed by SAPFIX were deemed, by developers, to correctly fix the failure. Of those deemed correct, about half were landed 'as is', and half were modified. Of those that were modified, about half were edited by the developers, while half were simply reviewed by the developer only after they had already fixed the bug themselves. Of the (approximately) half of all proposed fixes that were *not* ultimately landed into the code base, about half were deemed incorrect by developers (would have side effects or failed to tackle the true causes). For the remaining half that were not landed, the proposed fix was simply abandoned (after 7 days with no response from the developer).

Table 3.3 reports the results of attempting to revert developers' Diffs, where these may have affected on-going testing if not removed (or masked) in the master build. During its first three months of deployment, SAPFIX attempted to revert 18 Diffs (14 fully and 4 partially), where these Diffs contained high firing crashes, that could not be fixed by the templates or mutation-based fixing approaches. These Diff revert recommendations were all declined by developers (and not included in Table 3.1); it seems developers are (perhaps understandably) unwilling to simply revert their hard work. Nevertheless, anecdotally, we observe that the SAPFIX bot's suggestion of reversion did appear to engage developers with the urgency of providing a fix

3.3.1 Can SAPFIX fix pre-existing crashes?

The standard deployment mode, for which SAPFIX was designed (and is currently deployed), attempts to fix newly arising failures (crashes) as they are submitted in Diffs and detected as buggy by Sapienz. For this use-case, the developer has recent relevant context on the changes relating to the fix. Such relevancy has proved pivotal to the successful deployment (and human fix rates) for both Infer and Sapienz, as explained elsewhere [132]. Nevertheless, as a stretch goal we also experimented with targeting SAPFIX at pre-existing crashes that had reached production because Sapienz had failed to detect them (we are still working on the development of Sapienz [6], but no testing technology can be expected to stop *every* failure).

For pre-existing crashes, the developer reviewing the fixes proposed by SAPFIX has less context on the code and fix proposed. We split the results into two broad categories: long-standing (more than 3 months, the width of the Sapienz triage window [6]) and recent (first seen in the last 3 months, so potentially triagable by Sapienz, but missed by it). These long-standing crashes are also those for which the developer would be likely to have the *least* context, so it would be informative to see how many were landed by developers.

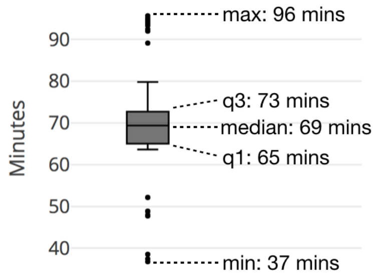


Figure 3.3: SAPFIX runtime: the time that SapFix requires to publish a fix.

In both cases (recent and long-standing) we cannot use precise localisation, since we do not have available Sapienz triage data. SAPFIX's *template_fix* and *mutation_fix* strategies rely on a blamed line to produce candidate fixes. However, for the pre-existing crashes that we target here, we do have access to multiple stack traces. Therefore, in this mode of deployment SAPFIX identifies the longest common path across 200 sampled stack traces, starting from the top of the stacks. The bottom-most line of this common path that is inside our codebase (not library or framework code) becomes the blamed line. Since this blamed line does not correspond to an identified Diff, SAPFIX instead uses a standard default approach, used across the company, to identify the developer to whom we should report the issue detected.

Table 3.2 presents the results for these fix candidates. These results are for a three day deployment window only. After three days, we switched off this experiment to avoid unnecessarily spamming our developers with multiple fix candidates from this comparatively untried-and-tested mode of deployment. In total, over the three days, SAPFIX constructed 946 candidates, of which it reported 195 to developers; 117 for recent crashes and 78 for long-standing crashes.

Had we left the experiment running longer, the proportion of landed fixes could only have increased, so we were encouraged that approximately 15% were landed within this three day period (which included a weekend; typically a quiet period for developer activity at Facebook [99]). Also, interestingly, we observed that the proportion of fixes landed was not notably different between recent crashes and longer-standing crashes. This observation gave us hope that we may ultimately be able to deploy this technology to track down and fix longer-standing crashes that developers tend to find harder to fix.

3.3.2 Timing issues

Figure 3.3 presents a time-to-fix box plot (from when SAPFIX is first notified of a need to fix, to the publication of a proposed fix to a developer). The median time from fault detection to fix publication to a developer is approximately one hour. More specifically, as shown in Figure 3.3, the median is 69 minutes, with a relatively tight inter-quartile range (65-73 minutes) and a worst case approximately 1.5 hours, and the fastest fix being reported to the developer 37 minutes after the crash was first detected.

As shown in Figure 3.3, the overall range of observed values is wide (37..96 minutes). This is because the timing figures are not only influenced by the computational complexity of fixing but also by the variations in workloads on the continuous integration and deployment system. Since SAPFIX is deployed in a highly parallel, asynchronous environment, the time from detection to publication can be influenced more by the demand on the system and the availability of computing resources than by the fix problem's inherent computational cost.

To create a candidate fix, once Sapienz has detected a crash, the maximum time allowed is approximately 1 hour. This limit arises because the workflow that detects that Sapienz has triaged a crash runs approximately once every 30 minutes. To create the fix typically takes approximately 30 minutes. Therefore, under normal overall system load, the overall expected time from crash detection in the master build used for debugging to publication of a fix to a developer through the Phabricator review system is approximately one hour.

3.3.3 Lessons Learned and Future Work

Our philosophy in deploying automated repair was to focus on industrial deployment, rather than further research. This philosophy has strongly influenced all of the decisions we took. We started with a collection of conceptually simple scalable techniques (static and dynamic analyses, template identification from developers' fixes, simple mutation rules, re-generation of tests, and so on). It was important that we already had evidence to indicate that each component we brought to bear was known to be deployable, to maximise the chance that, when combined, they would be able to achieve a simple, scalable and effective approach to repair. In particular, we have resisted the temptation to undertake research into the development of new details of the mutation-based repair algorithms, nor have we used advanced genetic programming, nor other techniques, such as constraint satisfaction, that may have posed challenges to us with deponent in our fully scaled continuous asynchronous integration and deployment environment. Despite their inherent appeal and potential, more research is needed to fully develop and extend these techniques to real-world deployment [184, 228, 247]. This is not to say that we believe these techniques are not scalable, but simply that our goal was to provide a successful and reliable infrastructural backbone for scaleable real-world deployment on which it would be possible *subsequently* to experiment with such more advanced research-led techniques. We were partly inspired by earlier work on automated repair, which demonstrated the potential for even relatively simple approaches to find reasonable repairs. For example, it has been known for some time that random search over a suitably-constrained (fault localised) search space, can be surprisingly effective at finding candidate repairs [184]. Indeed, several fixes reported in early work on repair were found in the very first generation [331] of the genetic programming execution, thereby pointing to the potential efficacy of pure random search. This observation was subsequently confirmed in later studies [184], that demonstrated that, when suitably constrained by fault localisation, and adequately

Table 3.1: Null Pointer Exception SapFix:"#P" is the total number of patches; "#M" is the number of mutation fix patches; "#G" is the number of template patches; "#Pass Tests" is the number of patches that passed all our tests; "#¬ Build" is the number of patches that failed to build; "#¬ Sap" is the number of patches that failed Sapienz testing; "#¬ Fix" is the number of patches that failed to fix the crash; "#¬ Pr." is the number of patches that were not published because the rule that produced them, was subsumed by one with a higher priority; "SapFix Land" specifies whether SapFix landed the patch into production; "Developers' Feedback" reports developers' insights. The table's time format is **MM.DD/HH:MM**. Note that on the 9th. August 2018 we updated our templates with better versions that covered more types of NPE fixes. We report the summaries of our results for both time periods and overall as well.

			Fix Strategy		#Pass Tests		Failed Patches			Dev Claims Correct			Dev says:		
	App	#P	#M	#G	#M	#G	#¬ Build	#¬ Sap	#¬ Fix	SapFix Land	Correct But Late	Correct Dev Land	Wrong Fix	Unkn.	Developers' Feedback
06.26/09:34	Facebook	3	0	3	0	3	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.06/09:29	Facebook	6	0	6	0	6	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.10/02:30	Facebook	4	0	4	0	4	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.11/06:04	Facebook	2	2	0	2	0	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.15/11:55	Facebook	2	2	0	2	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
07.16/03:42	Facebook	4	4	0	4	0	0	0	0	Yes	No	No	No	No	Macro: "superlike"
07.16/09:27	Instagram	1	1	0	1	0	0	0	0	No	No	No	Yes	No	Not sure about the side effects of the fix.
07.18/12:06	Facebook	4	0	4	0	4	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.20/02:39	Facebook	3	2	1	2	1	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
07.20/03:01	Facebook	1	1	0	1	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
07.20/03:33	Instagram	2	2	0	2	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
07.20/05:41	Instagram	1	1	0	1	0	0	0	0	No	Yes	No	No	No	Correct fix: "Oh this is cool! I didn't notice this diff until now. I have addressed the issue in this diff Dxxxxxxx but I wish I've seen this earlier. :-)"
07.20/05:50	Facebook	2	2	0	1	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.31/04:21	Facebook	7	0	7	0	7	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.02/01:49	Facebook	1	1	0	1	0	0	0	0	No	No	Yes	No	No	Correct fix, landed by the developer.
08.03/01:32	Facebook	1	1	0	1	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.03/01:56	Instagram	3	3	0	3	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
08.08/00:28	Facebook	2	2	0	2	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
Mutation Fix Overall		24	-	-	23	-	1	0	0	4	5	1	2	1	**Fix attempts with at least 1 passing test patch: 13/18**
Templates Overall		25	-	-	-	25	0	0	0	1	0	0	2	2	**Fix attempts with at least 1 passing test patch: 6/18**
Total NPE Fixes: 18		**49**	**24**	**25**	**48**		**1**	**0**	**0**	**5**	**5**	**1**	**4**	**3**	
On 08.09.2018 we updated our templates.															
08.20/07:46	Facebook	4	0	4	0	0	4	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
08.20/07:47	Facebook	3	2	1	1	0	2	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
08.20/07:47	Facebook	2	1	1	1	1	0	0	0	No	No	Yes	No	No	Correct fix, landed by developer
08.20/08:56	Facebook	1	0	1	0	1	0	0	0	No	No	No	Yes	No	"The fix might mask a race condition"
08.20/08:56	Facebook	1	0	1	0	1	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
08.21/02:46	Facebook	4	2	2	2	1	1	0	0	No	No	No	Yes	No	Fixing the crash, but not a reasonable fix.
08.22/02:51	Facebook	2	2	0	2	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.22/02:52	Facebook	2	0	2	0	2	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.05/09:04	Messenger	1	0	1	0	1	0	0	0	No	No	No	Yes	No	fix at the wrong line (this was a bug in sapfix :( )
09.05/09:04	Messenger	1	0	1	0	1	0	0	0	No	No	No	Yes	No	Wrongly triaged.
09.05/09:15	Messenger	2	2	0	2	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/09:12	Facebook	4	2	2	2	2	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
09.07/09:12	Instagram	1	0	1	0	1	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/09:12	Messenger	2	1	1	1	0	1	0	0	No	No	Yes	No	No	Correct fix, landed by developer.
09.07/09:13	Messenger	3	2	1	1	1	0	0	1	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/10:20	Instagram	1	1	0	1	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/09:29	Facebook	9	0	9	0	6	3	0	0	Yes	No	No	No	No	Macro: image with killing bugs
09.08/09:29	Messenger	1	0	1	0	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/09:29	WorkPlace	2	1	1	1	1	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.08/10:07	Facebook	1	1	0	1	0	0	0	0	No	No	Yes	No	No	Correct fix, landed by developer.
09.08/10:17	Facebook	1	1	0	1	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/10:21	Messenger	2	0	2	0	2	0	0	0	No	No	No	No	Yes	well this is cool (abandoned b/c not reviewed in time)
09.09/09:12	WorkChat	2	1	1	1	0	1	0	0	No	No	Yes	No	No	Correct fix, landed by the developer.
09.09/09:12	Facebook	6	2	4	2	3	1	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.11/01:37	Instagram	4	3	1	0	1	0	3	0	Yes	No	No	No	No	Macro: "whatatimetobealive3"
09.18/14:12	Facebook	3	3	0	2	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.18/14:13	Instagram	4	2	2	0	1	1	2	0	No	No	Yes	No	No	Correct fix, landed by the developer.
09.18/14:14	WorkPlace	3	3	0	1	0	1	1	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.18/14:14	Instagram	4	4	0	2	0	0	2	0	No	No	No	Yes	No	"Pretty sure this isn't the cause of the crash (Which is already fixed)"
09.18/14:14	Facebook	2	1	1	1	0	1	0	0	No	No	No	Yes	No	Wrongly triaged.
09.18/14:15	Facebook	5	2	3	1	3	0	1	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
09.18/14:16	Messenger	6	1	5	1	4	1	0	0	No	No	No	No	Yes	"We do want to crash because we wanna know when it can be null."
09.18/14:16	Facebook	2	2	0	1	0	0	1	0	No	No	No	Yes	No	"This isn't the right fix at all, but it is really cool :)"
09.18/14:15	WorkChat	5	1	4	1	4	0	0	0	No	No	No	Yes	No	Rejected without comments.
09.18/14:17	Facebook	7	4	3	4	3	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.18/15:16	FBLite	2	1	1	1	1	0	0	0	Yes	No	No	No	No	"lg2m :o"
09.18/15:46	Facebook	3	3	0	3	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.19/10:42	WorkPlace	5	2	3	1	2	2	0	0	Yes	No	No	No	No	"It would be nice if the bot would also add '@Nullable'" "to ImageOptions. Probably hard to do though :)"
09.19/14:00	Facebook	3	3	0	2	0	0	1	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
Mutation Fix Overall		56	-	-	40	-	4	11	1	5	1	3	6	7	**Fix attempts with at least 1 passing test patch: 27/39**
Templates Overall		60	-	-	-	43	17	0	0	4	1	2	4	6	**Fix attempts with at least 1 passing test patch: 22/39**
Total NPE Fixes: 39		**116**	**56**	**60**	**83**		**21**	**11**	**1**	**9**	**2**	**5**	**10**	**13**	
Results for our entire data set.															
Mutation Fix Overall		80	-	-	63	-	5	11	1	9	6	4	8	8	**Fix attempts with at least 1 passing test patch: 40/57**
Templates Overall		85	-	-	-	68	17	0	0	5	1	2	6	8	**Fix attempts with at least 1 passing test patch: 28/57**
Total NPE Fixes: 57		**165**	**80**	**85**	**131**		**22**	**11**	**1**	**14**	**7**	**6**	**14**	**16**	
%		**100**	**49**	**51**	**80**		**13**	**6**	**1**	**25**	**12**	**11**	**24**	**28**	
Total Correct Fixes(%)											**27/57(48%)**				

Table 3.2: SapFix experiment results on pre-existing crashes where we lack Sapienz triage data. "#C" is the number of crashes. The other columns are the same as those in Table 3.1.

		Strategy			Failed Patches				Developer says:		
Crash Type	#P	#M	#G	#Pass Tests	#¬ Build	#¬ Sap	#¬ Pr.	#C	SapFix Land	Wrong	Unkn.
Recent Crashes	**547**	288	259	213	166	65	103	**117**	16	35	66
%	**100**	53	47	39	30	12	19	**100**	14	29	56
Longstanding Crashes	**399**	230	169	139	132	39	89	**78**	13	24	41
%	**100**	58	42	35	33	10	22	**100**	17	30	53
Total	**946**	518	428	352	298	104	192	**195**	29	59	107
%	**100**	55	45	37	31	11	21	**100**	15	29	56

Table 3.3: SapFix results: High Firing, Non Null Pointer Exception Fixes. "#T" is the total number of patches; "#M" is the number of patches that the mutation fix strategy produced; "#G" is the number of template patches produced; "#Pass Tests" is the number of patches that passed all our tests; "$¬ Build" is the number of patches that failed to build; "#¬ Sap" is the number of patches that failed Sapienz testing; "#¬ Fix" is the number of patches that failed to fix the crash; "#¬ Pr." is the number of patches that were not published because the rule that produced them, was subsumed by one with a higher priority; "SapFix Land" is specifies whether SapFix landed the patch in an entirely automatic way, without human intervention; "Developers Feedback" specifies developers insights about the fixes and whether or not the developers consider the fix to be correct.

			Strategy		Patch Status					Dev Claims Correct:				Dev:	Dev:	
Date	App	#T	#C	#P	#Pass Tests	#¬ Build	#¬ Sap	#¬ Fix	#¬ Pr.	SapFix Land	Correct But Late	Correct Dev Land	Total Correct	Wrong Fix	Unkn.	Developers Feedback
2018-09-19 05:48:17	FB4A	4	1	3	1	2	0	0	1	No	No	No		No	Yes	"No need to rollback this"
2018-09-10 10:34:13	FB4A	1	1	0	1	0	0	0	0	No	No	No		No	Yes	"SapFix's fix here appears to just be reverting my diff... is that supposed to happen? What else can I do other than reject this diff?"
2018-09-09 05:58:34	FB4A	4	1	3	1	2	0	0	1	No	No	No		No	Yes	
2018-09-09 05:58:27	FB4A	4	1	3	1	2	0	0	1	No	No	No		No	Yes	
2018-09-08 09:29:30	FB4A	4	1	3	1	0	0	0	3	No	No	No		Yes	No	Wrongly triaged.
2018-09-07 12:17:41	FB4A	1	1	0	1	0	0	0	0	No	No	No		No	Yes	"nope :) bug is already fixed, thanks AI but not today, humans forever :D"
2018-09-05 08:27:15	FB4A	2	0	2	1	0	0	0	1	No	No	No		Yes	No	Wrongly triaged
2018-09-05 08:27:10	FB4A	4	1	3	1	2	0	0	1	No	No	No		No	Yes	
2018-08-21 05:46:44	FB4A	4	1	3	1	1	0	0	2	No	No	No		No	Yes	
2018-09-09 09:12:50	FBLite	2	0	2	1	2	0	0	0	No	No	No		No	Yes	
2018-09-05 08:27:16	FBLite	3	1	2	0	3	0	0	0	No	No	No		No	Yes	
2018-09-20 01:27:56	Insta	4	1	3	1	1	0	0	2	No	No	No		No	Yes	
2018-09-20 00:35:16	Insta	1	0	1	1	0	0	0	0	No	No	No		Yes	No	Wrongly triaged.
2018-09-20 02:28:22	Mess	4	1	3	0	1	0	3	0	No	No	No		Yes	No	
2018-09-09 03:28:36	Mess	4	1	3	1	1	0	0	2	No	No	No		Yes	No	"Looks like the diff just reverts the diff, which is not the correct fix. I will dig into why the crash happens and apply a better fix."
2018-09-08 23:57:37	Mess	4	1	3	1	1	0	0	2	No	No	No		Yes	No	Wrongly triaged.
2018-09-08 05:01:40	Mess	1	0	1	1	0	0	0	0	No	No	No		No	Yes	
2018-09-07 23:57:42	WChat	4	1	3	1	0	0	0	3	No	No	No		No	Yes	
Total High Firing		**42**	**11**	**31**	**13**	**14**	**0**	**0**	**16**	**0 / 18**	**0/18**	**0/18**	**0/ 18**	**6/18**	**12/18**	
%										**0%**	**0%**	**0%**	**0%**	**33%**	**67%**	

guided by testing, repairs can be found in reasonable time, on relatively large systems (although not as large as those on which we report here).

Earlier work on Sapienz had fortunately led to scalable and sufficiently precise fault localisation, which contributed to the 75% fix rate for human developers, reported on elsewhere [6]. The existing deployment of Sapienz, together with these results from the literature gave us confidence that we could deploy a relatively simple end-to-end repair approach as a starting point. We also sought to re-use developer-defined patch templates as a starting point, knowing that the scientific literature demonstrated that this can work [72], but also in the firm belief that this would lead to more human-acceptable patches.

Finally, we were also motivated by more recent work on automated repair that has highlighted issues concerning weak oracles [257]. To ameliorate this problem, we use a combination of static and dynamic analysis to check, re-check, localise and identify the code that needs to change. We also use a combination of regeneration of search based tests with Sapienz, and human-written end-to-end tests, to provide a testing environment in which to check the repairs constructed by SAPFIX.

Humans still play the role of final gatekeeper with SAPFIX: no repair is landed into production without human oversight, so the repair system, although fully automated is, nevertheless, at this point merely a recommender system. with the developer controlling the ultimate decision as to whether the proposed repair is committed to the production code repository. This final human gatekeeper phase also provides us with insights from real-world developers' reactions when presented with automated repair candidates, on which we report.

We target Null Pointer Exceptions (NPEs) in the first instance because NPEs are such an important category of fault [136]. NPEs are also a highly prevalent fault category: Coelho *et al.* [70] analyze a set of 6000 Android stack traces that they extracted from more than 600 Android apps. They observe that more than 50% of the crashes are NPEs. This lower bound of 50% has been replicated for the top 1,000 android apps, using Sapienz automated testing [208] and we also found, at Facebook, that NPEs constitute at least 50% of the crashes triaged by Sapienz to developers. All of this empirical evidence pointed to NPEs denoting a natural high impact class of faults on which to direct our initial focus. NPEs also have the advantage, for automated repair, that fixes tend to be localised and small. As such, we anticipated a higher probability that mutation operators, combined with identification of fix patterns may lead to successful deployment.

Much remains to be done, but we believe our initial deployment has allowed us to garner some experience, insights and initial results that may be useful to other researchers and practitioners, which we summarise in the remainder of this section.

End-to-end automated repair can work at scale in industrial practice: We have existential proof that developers do accept some automated patches; approximately one quarter of our patches landed into production code and a further quarter were deemed correct but not landed, either because the developer tweaked the fix or because they had already fixed the crash themselves when they first saw the proposed fix.

This is encouraging evidence that Automated Repair can work in practice, at scale, with professional developers regularly accepting patches into production in continuous integration and deployment. Clearly much more research and development work is needed to fully optimise and develop automated repair in general. We certainly do not underestimate the challenges that lie ahead for the international research community dedicated to this research agenda. Nevertheless, our positive results mean that the question as to whether end-to-end automated repair *could work in industrial practice* is now answered, allowing the community to devote its full energy to tackling the many (exciting and impactful) open problems.

Developers are a useful final oracle: Automated oracles [31] will hopefully become more sophisticated and help further reduce developer effort and widen the remit of automated repair. However, we have found that developers make good final gatekeepers, giving us a route to deployment while we await advances in oracle automation. Work on automated oracles can best support this aspect of the repair agenda by seeking to reduce the final developer effort required.

Sometimes deletion (reverting) is useful: high firing crashes in a master build of the system, even if never ultimately deployed to customers, will block further testing, so deleting them can be useful. This is an important use-case where the previously observed apparent predilection of automated repair to simply delete code (or to mask a failure, rather than tackling the root cause) is a behaviour that we seek; it can re-enable testing in the presence of a high-firing crash. Therefore, although deletion should be an anti-pattern for automated repair more generally [308], it is deployable in this specific use-case. Indeed, such behaviour is sufficiently important that we have two reversion workflows, specifically crafted to deploy 'deletion'. However, more research is needed on the problem of finding the right code to delete without affecting subsequently-landed code modifications. Program slicing techniques [124, 336] might find in this repair-orientated problem, a new application domain. We also found that developers are resistant to Diff reversion (perhaps understandably). More work is therefore required on partial deletion and crash-masking, so that the effects of a crash can be suppressed while minimally affecting onward computation, not only nor even necessarily for end-user release, but also to support further testing.

Sociology: Developers may prefer to clone-and-own proposed fixes, rather than to simply land them (approximately one quarter of fixes deemed correct by engineers were, nevertheless, edited by them prior to landing). More work could be done on the sociology of automated repair; the interfaces between human and machine in repair.

Automated Explanations: Developers often showed a readiness and interest in communicating with the SAPFIX bot (even though they *knew* it to be a bot), as indicated by their comments and feedback. There is a significant, and as-yet untapped, potential for *dialog* between the automated repair tool and engineer. More work is needed on techniques for repair (and more generally program improvement [128, 247]) that *interact* with the developer to 'discuss' proposed changes. Sapienz uses an automated experimental framework [6] that seeks to scale up best practice empirical software engineering experimentation. This could be a starting point for automated experimentation to provide further justification and explanation of proposed code modifications; something we hope to explore in future work.

Combine static and dynamic analysis: We have found that both static and dynamic analysis are complementary and mutually re-enforcing more generally [132], but also here in the specific case of automated repair. More work is needed to find blended analyses [94] that target repair.

Root cause analysis: A significant concern that developers raise and the SAPFIX team shares, is that SAPFIX might simply remove the symptom rather than addressing the root cause of a failure. This is a topic of on-going research in the literature. Techniques for identifying root causes of failures and appropriate (automatable) remediation remains very pressing problem.

Side Effects: Without fully automated oracles, our ultimate defence against side effects remains, as it does with human fixes, the developers and reviewers of Diffs. Much more work is still needed on automated test design and automation of (strong) oracles so that we can have greater confidence that passing all tests makes it unlikely that some knock-on effect is caused by a patch.

3.4 Conclusions and Future Work

The SAPFIX system is now running in continuous integration and deployment. As a result there now exist production software systems, of tens of millions of lines of code, used by hundreds of millions of people around the globe every day for communication, networking and community building, that contain fixes that result from automated repair. Furthermore, the repaired faults were revealed by entirely automated test case design and execution. The repairs to these faults were also automatically tested and were finally accepted by expert human engineers into the production code. This is the first time that automated end-to-end repair has been deployed into production on industrial software systems in continuous integration and deployment. Much remains to be done. Our repairs aim to tackle the most prevalent (yet arguably also the most simple) bugs, fixable by small patches, comparatively easily checked by the final human gate-keeper. As such, our SAPFIX approach tackles few of the many interesting and exciting open research problems for automated repair. Nevertheless, the Rubicon has been crossed: the topic of Automated Repair now has unequivocally demonstrated industrial applicability. We share the lessons we learned from the deployment of SAPFIX in this chapter seeking to provide additional input to the development of this challenging but important research field from a practical industrial perspective. We hope that this adoption milestone will act as a spur to on-going and further research and development on Automated Program Repair, Automated Software Testing and Search Based Software Engineering.

Chapter 4

Monolingual Automated Software Transplantation

In this chapter we present our approach for monolingual automated software transplantation. We implemented our approach in a tool called μSCALPEL, capable of transplanting non-trivial software *organs* (features) from real world donor programs, into real world host programs. We report the results of an empirical study, in which we run μSCALPEL on 5 donor programs, and 3 host ones, by doing all the possible combinations between them. In total, the empirical study contains 15 transplants, executed 20 times each, for considering the stochastic nature of the genetic programming based approach that μSCALPEL implements. We also report 2 case studies, where we transplanted very useful functionalities, between popular, large real world software systems.

The overall monolingual software transplantation approach, the empirical study, and the H.264 transplantation case study are based on our ISSTA 2015 paper [30]. The related experiments were accepted at ISSTA15-AE[1], the artifact evaluation track at ISSTA 2015. The Kate case study (Section 4.11) is based on our SSBSE 2015 paper [30]. μSCALPEL and all our experiments are available open source at `http://crest.cs.ucl.ac.uk/autotransplantation`.

This chapter is structured as follows: Section 4.1 introduces the problem of monolingual software transplantation; Section 4.2 presents 2 motivating example of running μSCALPEL: a complete run of μSCALPEL on an laboratory program; and a run of μSCALPEL on one of the experiments in the empirical study; Section 4.3 formally defines the general problem of software transplantation; Section 4.4 presents our approach and algorithm for the software transplantation problem; Section 4.5 reports our two implementations for monolingual software transplantation; Section 4.6 details the research questions that we aim to answer in this chapter; Section 4.7 and Section 4.8 reports our monolingual transplantation empirical study; Section 4.9 reports our H.264 transplantation case study; Section 4.10 presents our Humies transplantation entry that received the gold medal at Humies 2016;

[1]`http://issta2015.cs.uoregon.edu/artifacts.php`

Section 4.11 reports 2 case studies of transplanting functionalities into Kate[2], a popular KDE based text editor; and finally, Section 4.12 concludes our monolingual software transplantation approach.

4.1 Introduction

Monolingual software transplantation is the problem of software transplantation, when the language of the *host* and *donor* systems is the same. This chapter presents our algorithm for automated software transplantation, based on a new kind of genetic programming, augmented by a novel form of program slicing that we call μTrans. μTrans uses both static analysis and testing: static analysis extracts an 'organ', an executable useful behaviour from a *donor*; while testing guides all phases of autotransplantation — identifying the organ in the donor, minimizing and placing the organ into an ice-box, and finally, specializing the organ for implantation into a target host. Our observation based slicing technique [40] grows the organ in isolation, in an ice-box, until that organ passes the isolation test suite, thus incorporating the functionality that the test suite exercises, and is desired to be transplanted in the host system. To increase the performance and functional coverage of the test suite, we introduce *in-situ* testing, a novel form of testing that constrains the input space of traditional testing to more closely approximate behaviourally relevant inputs.

We implemented our monolingual software transplantation approach in μSCALPEL, for the C programming language, and evaluated it in 18 different transplantation experiments, using six donor programs and four host programs. Our current evaluation consists into an empirical study, and 2 case studies. In the empirical study we used 5 donor programs and 3 host programs, as following: each and every donor is transplanted into each and every host. As a sanity check, we also tried the identity transplant: we transplanted one donor into itself. All transplantation experiments are repeated 20 times, as the underlying algorithm, which is based on genetic programming, is stochastic.

All the systems involved in our experiments are real-world systems in current use, ranging in size from 0.4–25k lines of code for the donor and 25–363k for the host. We validate the resulted postoperative host systems, under two dimensions: first we check whether the organs disturbed anything in the host system (regression testing), and second we check whether or not the organ is reachable, is executing and exhibits the desired behaviour when transplanted into the *host* (acceptance testing).

We report 2 case studies. The first one is the transplantation of an encoder for the H.264 video encoding standard from the x264 encoder into the VLC media player. We know this to be an useful transplant, since the maintainers of VLC are downstream users of x264. This encoder changes every couple of years. For VLC, x264 consists of a binary library and a wrapper that translates VLC's internal protocols into the x264 API. To keep up, the VLC community must periodically replace their x264 library, including functionality in it that VLC does not use, and update their wrapper.

[2] http://kate-editor.org

Across 39 updates of x264 over 11 years[3], updating VLC for the new version of x264 averaged 20 days of elapsed time. In contrast, we ran μSCALPEL against the entire x264 codebase, targeting its `x264_encoder_close` function as the organ entry point. Obviously, it does not make sense to transplant an organ into a host that already has that organ. Thus, we stripped any reference to x264 from VLC by building it with `autoconfig --enable-x264=no`. Under this setting, the x264 library is not linked and the preprocessor eliminates VLC's x264 wrapper. This setup allowed us to produce a postoperative version of VLC that dispenses with the wrapper and slices out unused portions of the x264 library, dropping 47% of the x264's source code. This case study demonstrates that software transplantation can be automated and can transplant real software functionality that is both practical and useful.

The second case study illustrates the way in which realistic, scalable, and useful real-world transplantation can be achieved using autotransplantation . The host system is Kate[4] , a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. We perform two automated transplantations. In the first one, we transplant call graph drawing ability from the GNU utility program Cflow, to augment the features of Kate with the ability to construct and display call graphs.

This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, μSCALPEL, the developer merely needs to identify the entry point of the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [253], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in the present paper, but believe that existing achievements in Search Based [125] and other [55] test data generation techniques will help us to ensure that this reliance is reasonable and practical.

Our second transplantation incorporates a pretty printer for C, which `Kate` only partially supports and which its users have requested. At the time of writing, we deployed a new version of `Kate` that incorporates these features.

Our experimental results provide empirical evidence that software transplantation is a realistic and viable. In 12 out of 15 experiments, involving 5 donors and 3 hosts, we successfully transplanted functionality that passes all regression and acceptance tests. Moreover, autotransplantation is useful: μSCALPEL autotransplanted a H.264 multi-threaded encoding feature from x264 into the VLC media player in only 26 hours (Section 4.9); and μSCALPEL autotransplanted call graph generation and C identation, two missing functionalities from a lightweight C IDE such as Kate, that were requested by its user, in less than 132 minutes on average.

[3]We identified these 39 updates by manually inspecting VLC's version history, so this number is a lower-bound.
[4]http://kate-editor.org

Transplantation is a long-standing development practice. Currently, software developers manually identify the organ, extract it, transform it so that it executes in the host, then validate the correctness of the postoperative host. Identifying a donor, an organ's entry point in that donor, a host, and an implantation point in that host are hard problems that are currently not addressed by our approach. We present ideas for automatization in these areas in Chapter 7. Stepping in after these problems have been solved, μSCALPEL automates the extraction of the organ, *i.e.* the identification of its lines from its entry point, and its implantation into the host. In addition to an organ's entry point, μSCALPEL also requires a test suite that captures that captures the interesting behaviour to be transplanted into the *host*. Given these inputs, μSCALPEL computes an *over-organ*, an over-approximate, context-insensitive slice over the donor's call graph, then implements a kind of observation based slicing [40] in GP to reduce the organ, transform it to execute in the host, and validate the postoperative host. Our approach automates the later stages of software transplantation, making it faster, less laborious, and cheaper. Automated transplantation promises other new opportunities for enhancing development practices. For example, *speculative transplantation* may allow the exploration of feature reuse across systems that are currently untried because of cost.

4.2 Motivating Examples

This section presents 2 running examples of μSCALPEL. Section 4.2.1 presents a pedagogical example run of our tool, on a toy transplant example, with all the involved processing steps, and complete listings for the host and donor source codes. This example provides a clear understanding of our workflow, of the required manual work, and of the way in which μSCALPEL transplants new functionality. The second example, presented in Section 4.2.2 provides a motivating example for one of the experiments in the empirical study: `IDCT` coefficients computation from `IDCT` donor, into `cflow` host program. This running example will be further used in exemplifying our approach in Section 4.4.

4.2.1 Artificial Complete Workflow of μSCALPEL

This section presents an artificial example that shows a complete walk-through of the monolingual software transplantation process to facilitate its understanding and to clarify the manual steps that it involves. All the listings presented here are produced by μSCALPEL. These listings are a simplified version of the outputs that μSCALPEL produced. The complete pedagogical example on which this section is based is available in the artifact that we published on our website[5].

Figure 4.1 lists the source code of the host and donor programs. The host program H (Figure 4.1a) simply reads an array of integers from its CLI in the variable `array`, with its length `l`. The donor program D (Figure 4.1b) similarly reads an array of integers from its arguments in the variable `array`, with its length in the variable `length`, but has an additional functionality that we seek to transplant in H: it computes the sum of the array at line 29, by calling the function `computeSum`. The

[5]`http://crest.cs.ucl.ac.uk/autotransplantation/downloads/artifact.html`

```
1   void main () {
2       int *array;
3       int l;
4       printf ("Array␣length:␣");
5       scanf ("%d", & l);
6       printf ("\n");
7       array = (int *) malloc (l * sizeof
            (int));
8       for (int i = 0; i < l; i++) {
9           printf ("Element␣%d:", i);
10          scanf ("%d", & array [i]);
11          printf ("\n");
12      }
13     /* IP */
14  }
```

(a) The host program.

```
1    /* OE */ int computeSum (int *array, int
         length) {
2       int sum;
3       sum = 0;
4       for (int i = 0; i < length; i++) {
5           sum += array[i];
6       }
7       return sum;
8   }
9
10  void processArray (int *array, int l) {
11      if (!array)
12          return;
13      for (int i = 0; i < l; i++) {
14          if (array[i] < 0) {
15              array[i] = 0;
16          }
17      }
18  }
19
20  int main (int argc, char **argv) {
21      int length;
22      int *array;
23      array = (int *) malloc (length * sizeof
            (int));
24      length = argc - 1;
25      for (int i = 0; i < length; i++) {
26          array[i] = atoi (argv[i + 1]);
27      }
28      processArray (array, length);
29      int sum = computeSum (array, length);
30      printf ("The␣sum␣of␣the␣array␣is:␣%d\n",
            sum);
31      return 0;
32  }
```

(b) The donor program.

Figure 4.1: The host and donor for monolingual software transplantation artificial running example.

donor program calls the function `processArray` at line 28, prior to compute the sum, that removes the negative elements of the array.

Our lightweight annotation system is shown in Figure 4.1. In the host program, the user of μSCALPEL adds a comment with the content: `/*IP*/` at line 13 in H, for stating the point where the call to the transplanted organ to be added. This point is used by μSCALPEL to compute the symbol table available in H. On D's side, the user annotates the function `computeSum` as the organ's entry point, by using the annotation `/*OE*/` at the beginning of its declaration on line 1. While deciding where to place them may be difficult, `OE` and `IP` are the only two annotations μSCALPEL requires.

Now, suppose that the user wants to transplant the functionality of computing the sum of all the *positive* elements in the array (*i.e.* by processing the array first), from D to H.

Organ extraction: the first stage of μSCALPEL consists in the static analysis that constructs the *organ* and the *vein* (V) by looking at the donor's call graph. To construct the vein, we first identify the backward paths on the call graph from the organ entry point (*i.e.* the function `computeSum`) until the donor's entry point that is the function `main` for C programs. Our assumption is that each and every of the veins correctly and completely initialize the environment in the donor for the execution of the organ. Thus, we can simply pick one of them uniformly at random.

```
computeSum() <int computeSum () at
     main.c:6>:
```

(a) The *Organ* `computeSum`. This organ does not call any other function.

```
main() <int main () at main.c:25>:
    malloc()
    atoi()
    processArray() <void processArray () at
         main.c:15>:
    computeSum() <int computeSum () at main.c:6>:
    printf()
```

(b) The backward paths, from the organ entry that reach the donor's `main` function. In this case we have only one *Vein*: `computeSum` ← `main`

Figure 4.2: The output of the cflow for the Donor program.

To identify the organ, we look on the forward call graph, starting from the organ entry function: all the functions that it calls. We extract all the code of the organ and its vein by context-insensitively traverse the donor's call graph and transitively including all the functions called by any function whose definition we reach, and is available in the input code. The organ and its vein represent the search space for the *genetic programming* (GP) process that guides the transplantation process.

We inline all the function calls up to the organ entry function, automatically annotate the statements, declarations and compound statements on V, and generate fresh variables, as described in the Section 4.4. Figure 4.3 shows the obtained organ and vein, as μSCALPEL directly outputs it.

We use *GNU cflow* to compute the call graph of D. Cflow generates two types of graphs: *direct graphs*, displaying *caller* — *callee* dependencies, and *reverse graph*, displaying *callee* — *caller* dependencies.

Figure 4.2 shows the call graphs that μSCALPEL constructs for D. Figure 4.2a shows the organ. These are all the functions that forms the organ. In the current example, this contains just the function `computeSum()`, since the organ entry does not call any other function. Figure 4.2b contains the backward paths on the call graph. Here, different functions on the same level, represent different paths in the call graph, having the left hand side common. The `computeSum` organ has only one backward path to main: `computeSum` ← `main` and thus we pick this path as our *vein* V.

Further, μSCALPEL extracts and processes the code of O and V from the call graphs to use it as the search space for GP in the *organ adaption* stage. Figure 4.3 shows the results. First, μSCALPEL inlines the functions on V: starting from `main`, μSCALPEL inlines all the internal functions (*i.e.* only `processArray` in this example that is identified by the location information `<void processArray ()at main.c:15>` in Figure 4.2b) that the donors call until the organ entry `computeSum`.

μSCALPEL uses the annotations at the beginning of the lines in Figure 4.3 to produce the source code of a GP individual. */*DECL: x */* represents a variable declaration; */*STM: x* represents a statement in the program; and */*COMPOUND: x* represents a compound statement. The number `x` is a number that uniquely identifies each declaration, statement, or compound statement. When GP selects a compound statement `x`, all the lines annotated with */*COMPOUND: x */* will appear in the source code of the individual. When GP selects a statement, μSCALPEL automatically adds all its dependency on

```
Core Organ: [
int computeSum (int *array, int length) {
    /* TARGET FUNCTION */
    int sum;
    sum = 0;
    for (int i = 0; i < length; i++) {
        sum += array[i];
    }
    return sum;
}
]

Vein {
    /* DECL: 0 */ int $_main_argc;
    /* STM: 0 */ $_main_argc = 0;
    /* DECL: 1 */ char **$_main_argv;
    /* STM: 1 */ $_main_argv = NULL;
    /* DECL: 2 */ int $_main_length;
    /* DECL: 3 */ int *$_main_array;
    /* STM: 2 */ $_main_array = (int *) malloc ($_main_length * sizeof (int));
    /* STM: 3 */ $_main_length = $_main_argc - 1;
    /* COMPOUND: 0 */ for (int i = 0; i < $_main_length; i++) {
        /* STM: 4 */ $_main_array[i] = atoi ($_main_argv[i + 1]);
        /* COMPOUND: 0 */}
    /* DECL: 4 */ int *$_processArray_array1 = $_main_array;
    /* DECL: 5 */ int $_processArray_l1 = $_main_length;
    /* DECL: 6 */ void $ABSTRETVAL_ret_processArray1;
    //Mappings for function: processArray ($_main_array, $_main_length);
    /* COMPOUND: 1 */ if (!$_processArray_array1) {
        /* COMPOUND: 1 */ goto LABEL_processArray1;
        /* COMPOUND: 1 */}
    /* COMPOUND: 2 */ for (int i = 0; i < $_processArray_l1; i++) {
        /* COMPOUND: 3 */ if ($_processArray_array1[i] < 0) {
            /* STM: 5 */ $_processArray_array1[i] = 0;
            /* COMPOUND: 3 */}
        /* COMPOUND: 2 */}
    /* COMPOUND: 1 */ LABEL_processArray1 : if (0) {
    /* COMPOUND: 1 */ }
    /* DECL: 7 */ int *$_computeSum_array1 = $_main_array;
    /* DECL: 8 */ int $_computeSum_length1 = $_main_length;
    /* RETURN_MARKER */ return computeSum ($_computeSum_array1, $_computeSum_length1);
}
```

Figure 4.3: The processed donor program. This figure captures the automatically generated annotations that are used in the GP stage.

declarations. When GP selects a statement that affect the control flow of the program, such as `break`, μSCALPEL automatically adds the compound statement in which it appeared.

To capture the dependencies of a declaration (in its initialization) or statement on the declarations of the variables that it uses and the dependencies of a control flow statement on its compound statement, μSCALPEL constructs the *Organ Dependency List* (ODL) that we show in Figure 4.5c. μSCALPEL records the dependencies in the ODL, adds the annotations in Figure 4.3, and inlines the function calls on V by traversing the *abstract syntax tree* (AST) of D.

Figure 4.5 shows the outputs of the organ extraction stage that we construct in the AST traversal: the array of statements and compound statements (Figure 4.5a); the mapping space for the variables in the organ and its vein to variables in the host (Figure 4.5b); and the organ dependency list (Figure 4.5c). To construct the mapping space, μSCALPEL simply uses the types annotations: for each and every variable in O and V, μSCALPEL adds as candidate mappings all the variables in H, available at the implantation point, where the type annotations matches. `0` in Figure 4.5b means to keep the initial

```
START_TEST(test_1) {
    int myArray[5] = { 10, -20, 5, 15};
    int TransplantResult = GRAFT_INTERFACE( myArray, 4);
    if(TransplantResult != 10){
            ck_abort();
    }
}
END_TEST
```

Figure 4.4: The ice box test suite. One example test case.

variables names in D. The purpose of the variable mapping is to connect the program status of H to the one of O and V.

Organ reduction and adaption the second stage of μSCALPEL uses genetic programming (GP) to transplant, reduce, and adapt the organ. The inputs of the GP are the results of the static analysis stage, that we list in Figure 4.5 and that we discussed above. μSCALPEL also requires the organ test suite for specifying the semantics of the feature that guides the GP process. Different organ test suites lead to different features to be transplanted. The test case in Figure 4.4 captures the current desired feature in this running example: computing the sum across all the *positive* elements in the input array. To implement this functionality, GP transplant the code of the function `processArray` One can imagine different functionalities that might be obtained from this code such as computing the sum across *all* the numbers of the array that would also require the code of the function `processArray`

The GP algorithm searches on the organ statement array and on the organ matching table. On the organ statement array, GP selects which statements to transplant in H. On the organ matching table, GP selects which mapping it should use for every variables in O and V. μSCALPEL adds the variable declarations automatically and only when they are needed, by using the *Organ Dependency List* (ODL). Similarly, μSCALPEL automatically adds the compound statements sourounding a control flow statement (such as `break`) by using the ODL. The closing and opening brackets of a compound statement are added together with the compound statement, by using the same automatic generated annotation (*e.g.* `/*COMPOUND 0 */` in Figure 4.5). We pretty print the source code before all of our analysis, for being sure that a line contains exactly one statement, and for the structure of open / close brackets in the case of the compound statements.

In our current implementation we use '`check`'[6] unit tests framework for C programs. We use its output in computing the fitness of an individual. In respect to the output format of `check`, the user can design the organ test suite in any way.

We describe in detail our fitness function in Section 4.4.2. At a high level, for each of the test cases our fitness function has three equally weighted fitness components: compilation, executing the test case without crashing, and producing the correct output. If an individual does not compile, we do not further execute the test suite and simply assign the fitness value as 0.

[6]`http://check.sourceforge.net`

```
0: /*STM 0*/
1: /*STM 1*/
2: /*STM 2*/
3: /*STM 3*/
4:/*COMPUND 0*/
5: /*STM 4*/
4:/*COMPUND 1*/
4:/*COMPUND 2*/
4:/*COMPUND 3*/
5: /*STM 5*/
```

(a) The array of statements.

```
/*DECL 0*/: $_host_l, 0
/*DECL 1*/: $_host_array, 0
/*DECL 2*/: $_host_l, 0
/*DECL 3*/: $_host_array, 0
/*DECL 4*/: $_host_l, 0
/*DECL 5*/: $_host_array, 0
/*DECL 6*/: 0
/*DECL 7*/: $_host_array, 0
/*DECL 8*/: $_host_l, 0
```

(b) The possible mapping space between host and donor.

```
STM 0 → DECL 0;
STM 1 → DECL 1;
STM 2 → DECL 2, DECL 3;
STM 3: → DECL 2, DECL 0;
COMPOUND 0: → DECL 2;
STM 4: → DECL 3, COMPOUND 0,
     DECL 1;
DECL 5: → DECL 2;
COMPOUND 1: → DECL 4;
COMPOUND 2: → DECL 5;
COMPOUND 3: → DECL 4,
     COMPOUND 2;
STM 5: → DECL 5, COMPOUND 2;
DECL 7: → DECL 3;
DECL 8: → DECL 2;
RETURN_MARKER: → DECL 7, DECL
     8;
```

(c) The Organ Dependency List (ODL) that records the dependencies of statements on variable declarations and the dependencies of control flow statements on the compound statements that contain them.

Figure 4.5: The inputs of the GP stage.

```
MAPPINGS:
[
   DECL 4: $_host_array;
   DECL 5: $_host_l;
   DECL 7: $_host_array
   DECL 8: $_host_l
]
LOCS: [ COMPOUND 2, COMPOUND 3, STM 5, RETURN_MARKER]
FITNESS: [ 1.05 ]
```

Figure 4.6: The chromosome of the successful individual. The rest of mappings have no influence in this individual, since there are no statements involving them. RETURN_MARKER appears in all the individuals, as this is the organ entry point.

At every iteration GP selects with 50% chances to do a mutation or a crossover. In the case of mutation, GP selects with 50% chances to mutate a mapping or a selected LOC, as described in Section 4.4.

The chromosome of the successful individual is shown in Figure 4.6. In this case GP selected only the line with position 7. This line just divides the sum of the array, with the length of it for obtaining the mean. The value of sum in the example test case represents the sum of the array, while the asserted output is the square of the mean of the array. Obviously, there are possible other successful chromosome. In this example solution, the individual is not using the array, but only its sum and length. Another successful individual might consider just the values in the array, and then compute the mean with code from D.

Finally, Figure 4.7 reports the result of the transplantation, as μSCALPEL produces and implants in the host. This represents the instantiation of the chromosome in Figure 4.6.

```
int computeSum (int *array, int length) {
    int sum;
    sum = 0;
    for (int i = 0; i < length; i++) {
        sum += array[i];
    }
    return sum;
}

int sumOrgan(int $_host_l, int *$_host_array) {
    int *$_processArray_array1 = $_host_array;
    int $_processArray_l1 = $_host_l;
    for (int i = 0; i < $_host_l; i++) {
        if ($_host_array[i] < 0) {
            $_host_array[i] = 0;
        }
    }
    return computeSum ($_host_array, $_host_l);
}
```

Figure 4.7: The successful individual instantiated.

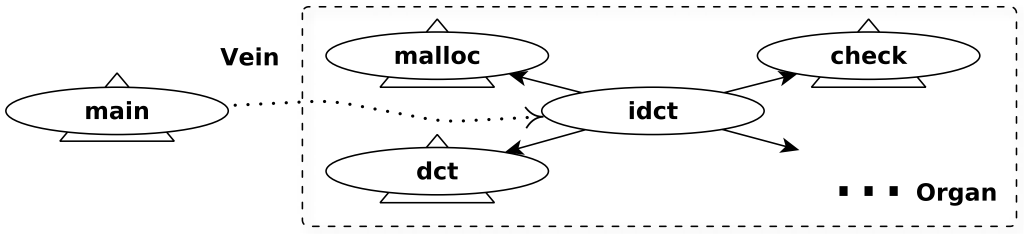


Figure 4.8: The call graph identifying the vein and the organ. The wavy line represents the vein; `main` is the entry point of the donor, an audio streaming client; `idct` is the organ's entry, and the rest of the nodes are on the forward slice. A triangle behind a node represents a subgraphs reachable from that node.

4.2.2 Running Example for Our Approach

For us, an organ is code whose functionality we wish to reuse by copying it to and adapting it for a new host, and a user annotates with `/* OE */`. One such organ is the function `idct` in Figure 4.8, which computes the inverse discrete cosine transform coefficients for an array of integers. Although there is little reason to move `idct` into cflow, we move `idct` from the donor, an audio streaming client, to cflow to illustrate our μSCALPEL works[7]. To mark `idct` as our organ, we change line 36 in file `dct.c` to `short *idct (int *idata){ /* OE */`.

Given a test suite, a donor with an annotated organ entry point, and a host with an annotated implantation point, μTrans solves two problems: identifying an organ within the donor and mapping that organ's variables to the host's variables at the implantation point.

We use slicing to identify the organ and one of its veins (as shown in Section 4.2.1). Fusing the slices of the organ and of the vein produces an over-organ that conservatively over-approximates the actual organ. To be viable, automatic transplantation cannot use the over-organ. To solve this

[7]http://crest.cs.ucl.ac.uk/autotransplantation/downloads/Idct.tar.gz contains this example; for readability, we have renamed the variable names in this section.

problem, μSCALPEL builds an organ that includes only those portion of the over-organ needed to pass a user-supplied organ test suite. For example, "`fwrite (prev_samples, 2, num_samples, ofile);`" (`recv.c`:64) outputs an audio file. The organ test suite ignores this functionality, so μSCALPEL removes it.

An organ requires a specific execution environment. To make the required program state explicit in the organ's signature, we lift globals in the unmodified organ in the donor into its signature. The signature for our running example contains the following local and global variables: "`int N, double scale0, double scale, FILE *ifile, FILE *ofile, int *odata, ...`".

The implantation point in the host defines an execution environment. We choose `output.c`:342 as the implantation point because cflow is consulting an array of line numbers for the call graph, so an array is available to the organ. After annotation `output.c`:342 is "`static void tree_output(){ /* IP */`".

We reify the host environment as an interface that captures all variables, both local and global. Transplantation requires mapping the parameters of this host-side interface to the organ's parameters. We can consider the organ's signature and the graft interface as set of parameters whose elements are typed formals. Under this interpretation, transplantation fails if the parameter set of host's graft interface is not superset of the organ's signature. The problem then is to efficiently find a ordering of a subset of the parameters in the graft interface that matches the organ's signature that is correct under an organ's test suite. For this, μSCALPEL binds the variables in the organ's signature at the parameters of `graft_idct`, and synthesizes a call. For our example, this call is: `graft_idct(arrayOfDefLines, length, symbols, main_sym, i, outfile, num, ...)`

4.3 Problem Formulation

This section defines the software transplantation problem. Chapter 2 uses the formalism introduced here to discuss the related transplantation approach. First we define our terminology. Next, we define the transplantation problem by using Hoare triples. In practice we use testing to approximate the Hoare triples.

Terminology D is a donor program from which we seek to extract a software organ O, code in D that implements some functionality of interest, to transplant. The organ O has entry points or locations, which we call L_O. H is the host program into which we seek to transplant O at $\square$, the target location(s) in the host. From the start of D, there may be many paths of execution that reach one of O's entry points; there is at least one, or O is nonfunctional and not worth transplanting. These paths build the program environment, Γ_O, on which O's execution depends. For instance, Γ_O may contain initialised globals or populated global data structures. In keeping with the transplantation analogy, we call these paths veins. We assume that all of them correctly build Γ_O, pick one, and call it V.

Software transplantation moves code that captures functionality of interest, the organ O, from a program D (the donor) into another program H (the host) that lacks this functionality. Software

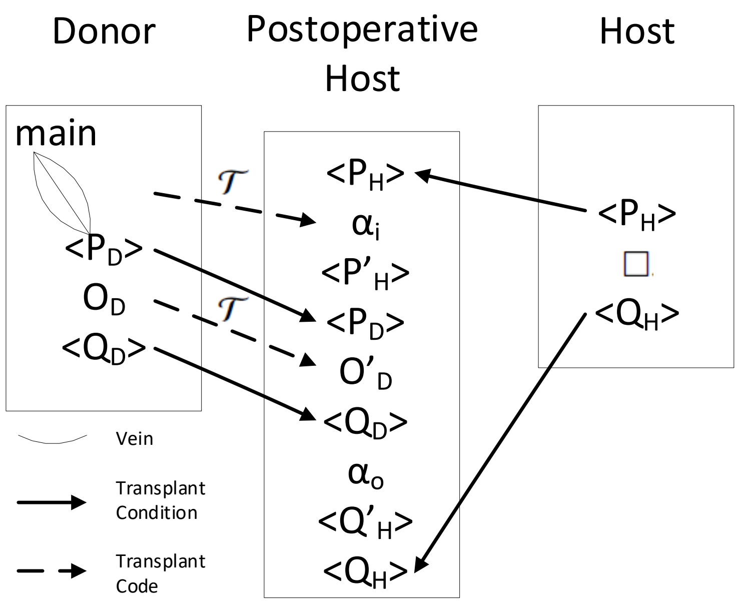


Figure 4.9: Automatic software transplantation formal definition.

transplantation comprises four subproblems: 1) to define O, 2) to find O_D, O as instantiated in D, 3) to find an implantation point for O_D in H, and 4) to adapt O_D to H. Defining O requires a description of the desired behaviour, such as a specification or test cases. When software is manually transplanted, usually via copy and paste, the developer informally knows the specification. Our approach for automated software transplantation requires a developer to manually identify O in D and an insertion point for O_D in H, then we automatically find O and transplant it into H: given an entry point to an organ (a function definition), and a test suite to underaproximate O's semantics, we initially overapproximates O, before pruning and adapting it for implantation into H via genetic programming.

Here, we explicitly capture the semantics of correct transplantation in Hoare triples. Since Hoare logic is undecidable [135], in practice we approximate the Hoare triples with testing. Starting from the host, whose need for functionality drives software transplantation, we have $P_H \square Q_H$ (where P_H is the precondition in H and Q_H is the postcondition in H). In the host H, the implantation point is a hole $\square$ that we want to fill with code the realizes some desired functionality, as O implements. At $\square$, H builds program states that satisfy the precondition P_D for O, or O would not be a candidate for filling $\square$. Assuming O already existed and had filled $\square$, its execution would produce states that add the desired functionality to H that is Q_H, the postcondition established by O.

Figure 4.9 shows our approach. Let σ_H be a state satisfying P_H at $\square$. At $\square$, H must supply a vector of actuals $\vec{a}$ for each of O_D's parameters in one of two ways: 1) H can directly supply $\vec{a}$ from σ_H or 2) H must construct $\vec{a}$. To construct $\vec{a}$, H can either transform σ_H or construct $\vec{a}$ from scratch. Let α_i denote adapter code injected into H to construct $\vec{a}$. To transform σ_H, α_i can either be constructed manually [287] or selected from a library of adaption components.

In D, we have $\langle P_D \rangle O_D \langle Q_D \rangle$. We assume that O_D is sufficiently correct that the developer wants to transplant it. More precisely we assume that all paths in D that reach O_D must contain the code that builds program state that satisfies P_D, O_D's weakest precondition. We call each of these paths in D a *vein*. By definition, a vein builds states that satisfy P_D, so it can construct an $\vec{a}$ from scratch and form the basis of α_i. Turning to the specification, we have $Q_{\alpha_i} \Rightarrow P_D$, *i.e.* α_i's postcondition must satisfy O_D's weakest precondition.

Organ transplantation involves inserting all statements from the organ O and V into the target site in the host. For the host, H, with target site $\square$, and the organ to be transplanted, O_D, the postoperative host is the middle figure in Figure 4.9. The function $\mathscr{T}$ is a the transplantation function that transplants and adapt O_D and V to the lexical context defined by $\square$: $\alpha_i = \mathscr{T}(V)$ and $O'_D = \mathscr{T}(O_D)$.

The output of O in the donor D may differ from its output in H'. For example, assume O sorts and outputs a set of numbers from a file and that D merely opens the file before invoking O, while H' opens and doubles the contents of the file before invoking O. We therefore need to equate observations that can be made of executions of O in the donor D to those in H'. We denote this equality $\simeq$. It can be thought of as a mapping from an oracle for O in the donor to an oracle for O in the postoperative host.

For a transplant to be successful, it needs to execute O'_D and to add the desired functionality:

$$\langle P_H \rangle \alpha_i \langle P'_H \rangle \Rightarrow P_D\text{: } \alpha_i \text{ correctly constructs } \Gamma_O\text{, the environment that the organ's execution requires} \tag{4.1}$$

$$\langle P_D \rangle O'_D \langle Q_D \rangle\text{: the organ behaves as expected} \tag{4.2}$$

We use testing to approximate Equation 4.1 and Equation 4.2:

Let T_A^D be the acceptance test suite for D that exercises the organ O. We say that an organ beneficiary passes acceptance testing if (and only if) all of the donor's acceptance test cases that exercise O yield observations that are equivalent under $\simeq$ to those same test cases adapted as necessary, by the function $\mathscr{T}$, for the organ beneficiary:

Definition 1 (Acceptance Testing)**.** The organ beneficiary H' *passes* acceptance testing iff $\forall t \in T_A^D \cdot D(t) \simeq H'(\tau(t))$.

Just as H's state at $\square$ may need α_i to supply O's inputs, O's output may need adaption so H can use it. For example, the adapter may need to transform O's output into suitable input for H's GUI. Let α_o denote this output adapter. For a transplant to be successful the organ must not disrupt the existing functionality of its host. Formally:

$$\langle Q_D \rangle \alpha_o \langle Q'_H \rangle \Rightarrow Q_H \tag{4.3}$$

assures this: after the execution of O, $\langle Q_H \rangle$ still holds. We use testing to underapproximate this constraint. To check Equation 4.3, we regression test the postoperative host against H's test suite.

Let T_R^H be the regression test suite for a host H. We say that the organ O'_D passes regression testing if (and only if) all regression test cases yield identical observations on the original host H and the postoperative host H':

Definition 2 (Regression Testing)**.** The organ beneficiary H' *passes* regression testing iff $\forall t \in T_R^H \cdot H(t) = H'(t)$.

To be successful, a transplanted organ must 1) not disrupt the existing functionality of its host (Equation 4.3) and 2) must actually execute and add the desired functionality to its host (Equation 4.1 and Equation 4.2). If 1) and 2) hold, we say that software organ transplantation has occurred:

Definition 3 (Software Organ Transplantion)**.** *Software organ transplantation* has occurred when $[\![H]\!] = [\![H']\!]_{\neg O} \wedge [\![D]\!]_O \simeq [\![H']\!]_O$, where $[\![P]\!]_O$, borrowed from denotation semantics [301], is the meaning of P restricted to the observable behaviors of O in P.

It is the responsibility of the programmer to select D, an entry point L_O to some functionality of interest in D, H, its target site $\square$, and to define $\simeq$ and an underapproximation of O through testing. It is the responsibility of autotransplantation to find V, O, and insert O into H at $\square$. An organ in the donor and its version after transplantation are type 4 clones, related by $\mathscr{T}$ and semantically equivalent under T_A^D.

Monolingual software transplantation is a straightforward implementation of Figure 4.9. $\mathscr{T}$ only moves code, without translation, as μScalpel operates only on the C language.

4.4 The μTrans Approach

Figure 4.10 shows the overall architecture of the μTrans approach to the software transplantation problem. Given an organ's entry point L_O and T_D^O, a test suite that exercises O's behaviour, μTrans progressively "evolves" O and V by adding code along a path (a vein) from donor start to L_O. The search space of possible organ veins is enormous, consisting of all combinations of all valid statements and variables of the desired functionality in the donor and host. To explore this search space efficiently, we separate our approach into two stages.

In the first, organ-extraction stage, we slice out an over-organ, a conservative over-approximation of the organ and one of its veins. The organ implements the functionality we wish to transplant; the vein builds and initializes an execution environment that the organ expects. In the second, implantation stage, we apply genetic programming, using the donor's test suite, reduce the over-organ and adapt it to execution environment at L_H, the insertion context in the host. The second stage implements observational slicing [40] in GP to simultaneously reduce the over-organ and explore mappings of the organ's parameters to variables in scope at the insertion point in the host. The second stage ends

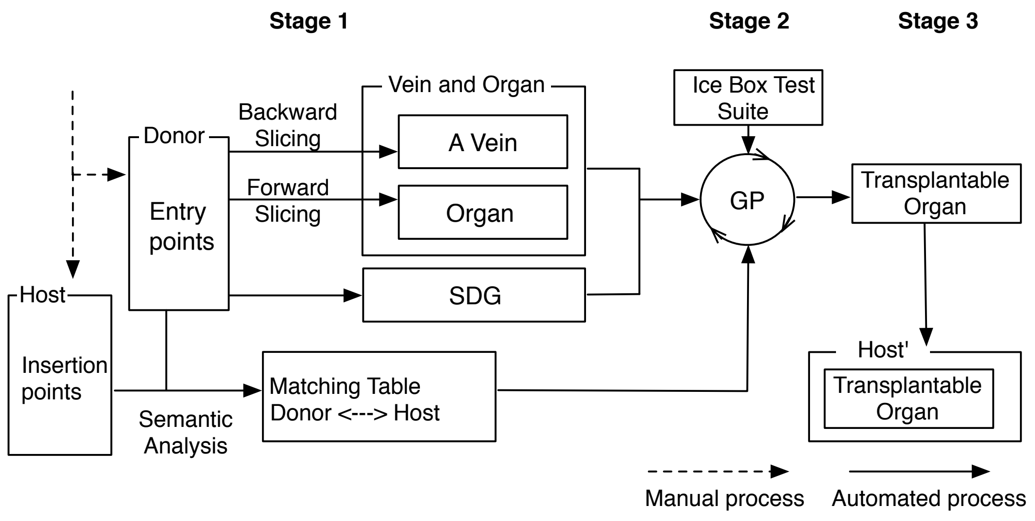


Figure 4.10: Overall architecture of μTrans; an SDG is a system dependency graph that the genetic programming phases uses to constrain its search space.

by implanting the organ into the host. Finally, we apply additional testing to validate the correctness of the postoperative host and to mitigate the risk of organ over-reduction that should be caught by our additional validation. This section explains the first and the second stages of μTrans in detail; Section 4.7 discusses its validation.

4.4.1 Stage 1: Organ Extraction

This stage takes the donor, annotated with L_O, the organ's entry point in the donor, and a test suite that exercises the organ's functionality. It produces two outputs for the second stage. First, it constructs an *over-organ*, a vein and an organ, that contains all the code in the donor that implements the organ, given the organ's entry with L_O. Second, it builds the donor's System Dependence Graph (SDG) which is used during GP to reject syntactically invalid offspring.

To produce the over-organ's vein and the organ below, we produce a conservative, over-approximate slice by traversing the donor's call graph; this traversal is context-insensitive: we include all functions called by any function we reach. An organ's vein is a path from a donor's start to L_O, organ's entry point. It contains all source code that constructs the organ's parameters. To create it, we backward slice the donor from L_O, traversing the call graph in reverse. Once we reach the donor's main, we prune the slice to retain only the shortest path, under the assumption that all paths to the organ are equivalent with respect to correctly constructing the organ's parameters. The organ's vein in the case of the running example (Section 4.2) is: `idct <- recv_packet <- transmit_packet <- send_packet <- main`. To construct the initial organ, we forward slice the donor from L_O. The forward slice for the idct organ (Section 4.2) is: `check, dct, malloc, cos, printf`

This over-organ is quite imprecise; we rely on the second stage to refine it to a small organ suitable for implantation. Investigating more precise techniques is future work.

Last, this phase outputs data and control dependency information. The GP process may generate many invalid programs, offspring that use undeclared variables or include only a syntactically invalid part of a compound statement, like an `if-else`. Both of these issues occurred in the file `recv.c` in `idct`, our running example. In `recv.c`, selecting `missing_seqno = prev_seqno + 1;` at line 55 triggered the inclusion of the declaration `static int prev_seqno;` at line 43. Another individual contained "`else if (nrOfLostPackets == 2)`" at the line 119. Tracking the data dependency ensures we include the declarations `nrOfLostPackets`, while the control ensures the inclusion of `if (nrOfLostPackets == 1)` (line 102), the predicate of the `else if` statement, since, in isolation, the statement violates C syntax.

4.4.2 Stage 2: Organ Reduction and Adaption

The second stage of μTrans has two steps. The first searches for bindings from the host's variable in scope at the implantation point to the organ's parameters. Some of these variables need to be created and initialized during implantation; others use existing host variables. In the former case, we use α-renaming [28] to avoid variable capture; in the latter, we use types to restrict the space of possible mappings.

We use Genetic Programming (GP) to evolve an organ and its vein from the over-organ produced by the first phase. To this end, we introduce *in-situ* unit testing. Traditional unit testing covers infeasible paths because it does not assume knowledge of how the unit will be used, *i.e.* constraints on its starting program states, and therefore executes paths that are infeasible *in-situ*, when the unit is embedded in a program. *In-situ* unit testing is a novel form of unit testing that starts from a valid program state rather than an arbitrary state. It loops over this state, modifying individuals to maximize coverage.

In-situ is a generally useful concept. In addition to its use in GP, it increased the rigor of our validation. Hosts tend to have large input spaces into which μTrans inserts alien code. Finding a path in the beneficiary to the transplanted organ can be difficult. *In-situ* unit testing allows us to leverage a single path to rigorously test whether the new functionality executes correctly in the host. During *in-situ* testing, any calls the unit makes to its enclosing environment, like its host in autotransplantation, are executed and not stubbed or mocked, as they would be in traditional unit testing [33].

μTrans thus realises a form of dynamic, observational slicing which removes redundant states from V while 1) ensuring that O and V remain compilable and executable, and still correctly construct the parameters that O needs to retain the correct behaviour as defined by T_D^O, and *in-situ* unit testing.

Figure 4.11 shows an example of the chromosome used in our GP. We inline all all functions, then map the over-organ's statements to an array; each array index uniquely identifies each statement. The chromosome of each individual has two parts: an host-to-organ map and a list of the indices in the over-organ that this individual includes.

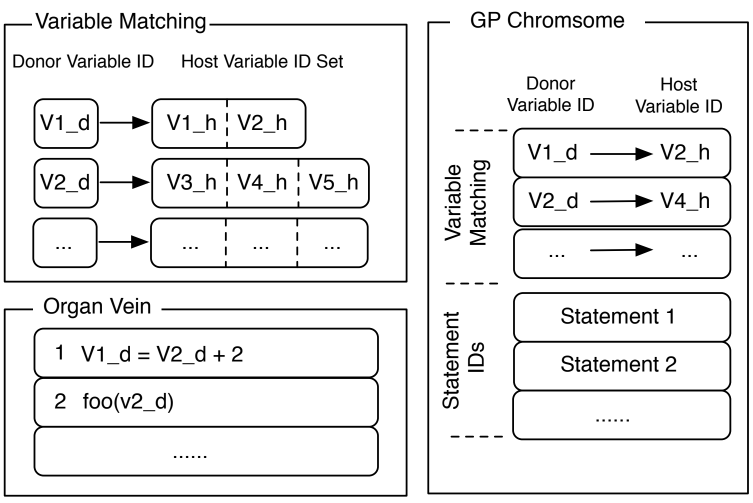


Figure 4.11: An example of μSCALPEL's GP chromosome.

Algorithm 7 Generate the initial population P; the function choose, uniformly at random, returns an element from a set.

```
Input  V, the organ vein; S_D, the donor symbol table; O_M, the host type-to-variable map; S_p, the size of population; v, statements in individual; m, mappings in individual.
1: $P := \emptyset$
2: for $i := 1$ to $S_p$ do
3:     $m, v := \emptyset, \emptyset$
4:     for all $s_d \in S_D$ do
5:         $s_h :=$ choose($O_M[s_d]$)
6:         $m := m \cup \{s_d \to s_h\}$
7:     $v := \{$ choose($V$) $\}$
8:     $P := P \cup \{(m, v)\}$
9: return $P$
```

Unlike conventional GP, which creates an initial population from individuals that contain multiple statements, μSCALPEL generates an initial population of individuals with just 1 statement, uniformly selected. μSCALPEL's underlying assumption is that our organs need very few of the statements in their donor. We also want to evolve small organs. Starting from one LOC gives μSCALPEL the possibility to find small solutions quickly.

Algorithm 7 shows the process of generating of the initial population. The organ's symbol table stores all feature-related variables used in donor at L_O. For each individual, Algorithm 7 first uniformly selects a type compatible binding from the host's variables in scope at the implantation point to each of the organ's parameters. We then uniformly select one statement from the organ, including its vein, and add it to the individual. The GP system records which statements have been selected and favours statements that have not yet been selected.

Search operators During GP, μTrans applies crossover with a probability of 0.5. μTrans defines two custom, crossover operators: fixed-two-points and uniform crossover. The fixed-two-points crossover is the standard fixed-point operator separately applied to the organ's map from host variables to organ parameters and the statement vector, restricted to each vector's centre point. The uniform crossover operator only produces one offspring, whose host to organ map is the crossover of its parents' and

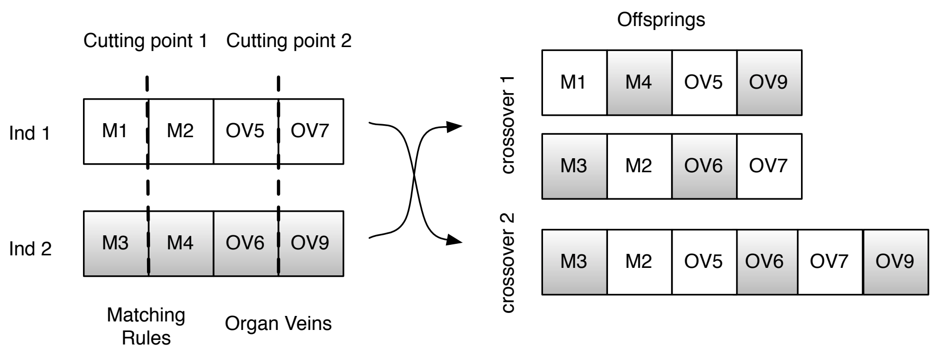


Figure 4.12: An example application of μTrans's two crossover operators.

whose V and O statements are the union of its parents. The use of union here is novel. Initially, we used conventional fixed-point crossover on organ and vein statements vectors, but convergence was too slow. Adopting union sped convergence, as desired. Figure 4.12 shows an example of applying the crossover operators on the two individuals on the left.

After crossover, one of the two mutation operators is applied with a probability of 0.5. The first operator uniformly replaces a binding in the organ's map from host variables to its parameters with a type compatible alternative. In our running example, say an individual currently maps the host variable `curs` to its `N_init` parameter. Since `curs` is not valid array length in `idct`, the individual fails the organ test suite. Say the remap operator chooses to remap `N_init`. Since its type is `int`, the remap operator selects a new variable from among the `int` variables in scope at the insertion point, which include `hit_eof`, `curs`, `tos`, `length`, . . . The correct mapping is `N_list` to `length`; if the remap operator selects it, the resulting individual will be more fit.

The second operator mutates the statements of the organ. First, it uniformly picks t, an offset into the organ's statement list. When adding or replacing, it first uniformly selects a index into the over-organ's statement array. To add, it inserts the selected statement at t in the organ's statement list; to replace, it overwrites the statement at t with the selected statement. In essence, the over-organ defines a large set of addition and replacement operations, one for each unique statement, weighted by the frequency of that statement's appearance in the over-organ. Figure 4.13 shows an example of applying μTrans's mutation operators.

At each generation, we select top 10% most fit individuals (*i.e.* elitism) and insert them into the new generation. We use tournament selection to select 60% of the population for reproduction. Parents must be compilable; if the proportion of possible parents is less than 60% of the population, Algorithm 7 generates new individuals. At the end of evolution, an organ that passes all the tests is selected uniformly at random and inserted into the host at H_l.

Fitness function: For the autotransplantation goal, a viable candidate must, at a minimum, compile. At the other extreme, a successful candidate passes all of the T_D^O, the developer-provided test suite

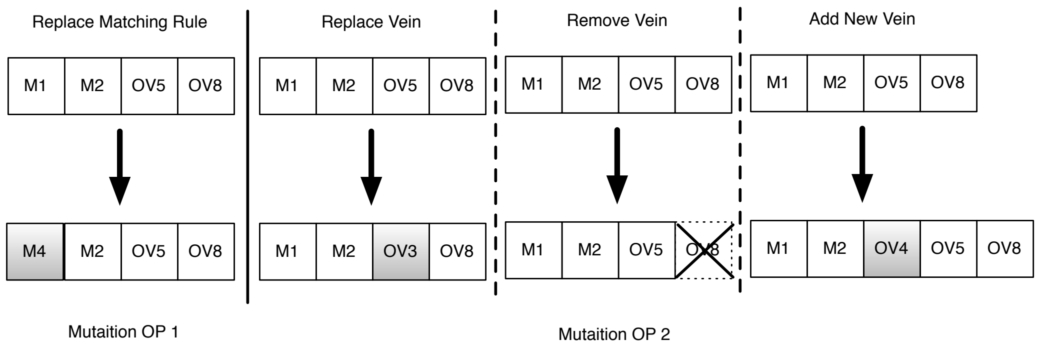


Figure 4.13: An example of two mutation operators.

that defines the functionality we seek to transplant. These poles form a continuum. In between fall those individuals who execute tests to termination, even if they fail. Our fitness function therefore contains three equally-weighted fitness components. The first checks whether the individual compiles properly. The second rewards an individual for executing test cases to termination without crashing and last rewards an individual for passing tests in T_D^O.

Let I_C be the set of individuals that can be compiled. Let T be the set of unit tests used in GP, TX_i and TP_i be the set of non-crashed tests and passed tests for the individual i respectively. Our fitness function follows:

$$fitness(i) = \begin{cases} \frac{1}{3}(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}) & i \in I_C \\ 0 & i \notin I_C \end{cases} \tag{4.4}$$

4.5 Implementation

μSCALPEL implements μTrans and is available open source at http://crest.cs.ucl.ac.uk/autotransplantation. This section describes the implementation of our tool that was initially implemented in TXL and C (Section 4.5.1) and later implemented in ROSE and C++ (Section 4.5.2) to tackle some limitations that we saw in the initial implementation. We discuss the limitations of μSCALPEL (and ROSIEμSCALPEL) in Section 4.5.3. Finally, Section 4.5.4 discuss the relationships between μSCALPEL and other code reuse techniques.

4.5.1 μSCALPEL

Initially we implemented μSCALPEL in TXL [71] and C. A TXL based implementation inherits the limitations of TXL, such as its stack limit which precludes parsing large programs (that leads to μSCALPEL not being able to scale to big programs) and its default C grammar's inability to properly handle preprocessor directives. As an optimisation we inline all the function calls in the organ. Inlining eases slice reduction, eliminating unneeded parameters, returns, and aliasing. For the construction of the call graphs, we use GNU cflow, and inherit its limitations related to function pointers. You can download μSCALPEL from http://crest.cs.ucl.ac.uk/autotransplantation.

μSCALPEL realizes μTrans and comprises 28k SLoCs, of which 16k is TXL, and 12k is C code. The TXL code does the program transformations required in monolingual software transplantation (eg. parsing the source code of the donor program; inlining function calls; adding annotation for declarations and statements such that the GP stage can replace them to the values that the GP algorithm selected; alpha renaming). We use TXL together with the information from GNU cflow to constructs the inputs for the GP stage: the over organ and the donor's System Dependence Graph. This implements the first stage of μTrans: *Organ Extraction* (Section 4.4.1). The C code implements the GP process and glues together the calls to the different TXL parsers, realising the second stage of μTrans: *Organ Reduction and Adaption* (Section 4.4.2).

4.5.2 ROSIEμSCALPEL

For handling the TXL scalability limitations that we discussed in Section 4.5.1, we completely reimplemented μSCALPEL, by using the ROSE compiler infrastructure (see Section 2.11.1). Since ROSE is implemented in C++ language, we reimplemented mostly from scratch μSCALPEL. All the TXL components where replaced with ROSE, while the GP algorithm was modified for handling the new ROSE's environment (data structures and languages). The results that we report in this thesis are about the ROSE implementation of μSCALPEL. Implemented in C++ and based on the ROSE compiler infrastructure (Section 2.11.1), the new implementation of μSCALPEL (that we call in this section ROSIEμSCALPEL to distinguish it from the TXL implementation) comprises 34k SLoCs. You can download μSCALPEL from `http://crest.cs.ucl.ac.uk/autotransplantation`.

Tackling TXL Limitations: ROSE enables us to be very scalable, by not being limited by the TXL's [71] size of files, and size of transformations limits. If TXL was not capable for example to parse a 30k LOCs file, ROSE does not have any limit in the sizes of the input files. Also, if a lot of transformations were required in the parse of a file, TXL was triggering an out of stack space error. In ROSE, we can do any number of transformations on a given file.

Another problem that we observed with TXL is the fact that the default C grammar cannot handle preprocessor directives. Because of this, we had either to automatically preprocess the source code, or to manually preprocess the parts of the preprocessor related code that was triggering errors in TXL parsings. Automatically preprocessing the entire source code was often generating files over the size limit of TXL in parsing. Because of this, we usually manually preprocessed the error triggering parts. In general, TXL's default C grammar is not able to parse the preprocessor instructions that break the syntactic correctness of C programs. For example, an instruction as: `#ifdef VAR_INT int #else double #endif x;` is triggering an error in TXL. ROSE does not have any of these problems, and is capable of parsing any preprocessor code.

Another problem that we observed with TXL is that it is not able to process more languages. Our initial μSCALPEL implementation is working just for C programs. If we want to handle a different programming language, then we require an additional grammar, and a totally different implementation of μSCALPEL, for any different grammar. ROSE uses a programming language independent abstract syntax tree (AST), such that our code is capable of running on all the languages supported by ROSE. Although our multilingual software transplantation work does not use ROSE, because of ROSE lacking front ends for the languages that we consider in Chapter 5, in future we could extend ROSIEμSCALPEL to be a more reliable multilingual software transplantation tool.

Thus, using ROSE, makes our autotransplantation tool, ROSIEμSCALPEL, far better than the previous TXL based implementation (μSCALPEL). We can summarize the advantages of ROSE as: support for the entire TXL part and much more; safe replace for the most of the current TXL based components, which contain a lot of bugs; completely removes risks for bugs related to grammar and TXL transformations; preprocessor support; annotations support; the implementation may become language independent; better support for unification; easier typedefs and structs matching. However, there are some disadvantages of using ROSE. Since ROSE is a C++ library, and the TXL based version of μSCALPEL is C based, for using ROSE we had to reimplement the most parts of μSCALPEL. However, we obtained a far scalable, robust, and better tool, than the TXL based version.

Challenges in Using ROSE: there were couple of challenges involved in making ROSE suitable for monolingual software transplantation and scaling it's at the sizes of real world programs. The biggest challenge were about our vein inlining approach, and entire project analyse. In theory ROSE supports whole-project analysis, by the AST merging mechanism. However, we identified some problems with the AST merging mechanism, which required us to reimplement it in a more robust way. The ROSE's builtin ability to generate call graphs was also working just for the files provided by the user at the compilation line, so not in an entire project. We reimplemented also the call graph generation.

ROSE is a compiler infrastructure, and thus is aimed to be used as a compiler. We use our generated translators as compilers, to replace the default compiler used in the Makefiles for the analysed projects (host and donor). Under these conditions, ROSE fixed up merge mechanism and call graph generations works just for all the files that are provided together at the compilation lines. For example, in: `muscalpel -o output.o file1.c file2.c` we can merge the AST's of `file1.c`, and `file2.c`. However, if `file3.c` is provided in a different compiler call in the Makefile, than ROSE will not be able to merge the ASTs.

For solving this problem, we decided to have a 2 compilation stage for ROSIEμSCALPEL. In the first compilation stage, we replace the compiler in the Makefile with ROSIEμSCALPEL–Stage1. In this stage, we output serialised versions of ASTs for all the source file in the input donor project. Like this, we can merge the ASTs for the entire project, rather than just the ASTs provided at a compilation line. All this ASTs enter in ROSIEμSCALPEL–Stage2. Here, we read the serialised ASTs, merge them together by using the fixed AST merge mechanism, and do the necessary transformations and analysis, as specified in Section 4.4, on the merged AST of the entire project.

Our concept for entire project AST merger is similar with the linking of the binary files. In ROSIEμSCALPEL–Stage2, we just link all the ASTs resulted from ROSIEμSCALPEL–Stage1, similarly with how a linker would link the object files resulted from the compilation stage. On the project's merged AST we do all our further analysis: first we generate the call graph of the donor program; then we context–insensitively slice the vein and the organ on this call graph (see Section 4.4); we inline the functions on the *vein*; extract the vein and the organ; construct the system dependency graph, and the donor–host matching table. This output enters in the genetic programming algorithm, which was reimplemented for handling ROSE–style data structures.

4.5.3 Limitations

To tackle the scalability and pre processor limitations of μSCALPEL, we implemented ROSIEμSCALPEL (Section 4.5.2). Other limitations of μSCALPEL are mainly related to the suitability of the implantation point in the donor for the organ (α_i in Section 4.3); deep integration between the transplanted organ and the host that is dependent on the input and output adapters (α_o in Section 4.3), and to the execution time of the GP stage in the case of real world organs that could potentially consist in hundred of thousands or even millions lines of code.

μSCALPEL first extracts the organ whose functionality is captured by the organ test suite. To integrate it into the host environment, μSCALPEL uses the organ's vein to connect it at the program state available at the implantation point. At a high level it does so by matching variables available at the implantation point with variables from the over-organ. The vein executes the rest of the computations that prepare the execution environment of the host for the execution of the organ, starting from the program state at the implantation point. The vein is our input adapter (α_i) in Figure 4.9. It is not always possible to find such vein in the donor program that can correctly initialise the host's environment starting from the program status at the implantation point. When such input adapter does not exist, the transplant fails.

For example, consider a simple case when we want to transplant the functionality of removing an element from an array. The donor program gets the array and the element to remove from its CLI inputs and thus, it doesn't contain the logic to read the element to remove. If the host program does not contain the element to remove at the implantation point, it is not generally possible to define an adapter from the host's program status to the organ, as we do not have access to which element to remove (unless the implantation point is in the parsing of the CLI arguments in the host). In such cases we have two options to make the transplant feasible:

- Manually provide the missing state in the host at the implantation point. For example, we could manually add code that reads the element to remove into the host at the implantation point.
- Execute two different transplants. We might be able to identify another donor program (or even another organ in the same donor program) that has the functionality of constructing the missing state from the host's implantation point. In the above example, we might find another organ that reads an input value from either CLI or GUI if the host is a GUI application. That value further could be used as the element to remove from the array. Chapter 6 discuss future work

ideas to use multi donor transplants. Currently we could try to apply μSCALPEL twice in such cases.

The output adapter α_o (Figure 4.9) realises the integration between the organ's output and the host. μSCALPEL does not explicitly construct such output adapter. μSCALPEL relies on the organ containing the required adapter in its initial source code. The organ's test suite validates that such adapter is present as without it the test case would fail. Unfortunately such adapter does not always exist. For example let's say that for our Kate call graph transplant (Section 4.11) we would like to additionally have the capability of using the call graph to interactively navigate through a program. Such functionality is not present in the donor *GNU cflow*. As in the case of the input adapters, here we can either manually construct such functionality once the transplant is done, or execute another transplant if we were to find a donor to contain the functionality of navigating through source code.

In the case of our x264 to VLC transplant(Section 4.9), μSCALPEL needed 26 hours to extract and transplant the H.264 encoding organ from x264 into VLC. In this case the organ had 23k lines of code. The vast majority of the time here was spent in the GP process that takes longer for bigger programs. If we want to keep transplantation feasible for big real world donor programs, that can consist in hundred of thousands , or even millions of lines of code, we need to optimise the μSCALPEL's execution time. A simple solution here would be to parallelise μSCALPEL's GP algorithm by using parallel genetic programming [14].

Finally, although μSCALPEL does not require too many inputs, the quality and feasibility of a transplant are highly dependant on the quality of its three main inputs: implantation point in the host, organ's entry point in the donor, and the organ's test suite. The quality of the organ test suite is highly correlated with the quality of the transplant. We discuss in Chapter 6 future work ideas improve organ's test suite quality and to even try to automatically generate it. The feasibility of the transplant is highly correlated with the organ entry point and the implantation point. The users of μSCALPEL need to carefully look at the code of the host and donor program to identify good places for these two annotations. Future work might allow our users to explore multiple candidate implantation points and organ entries.

4.5.4 μSCALPEL and Code Reuse

Automated software transplantation is a form of code reuse. Here we discuss μSCALPEL in relation with software clones [166], library, and direct usage of the organ in the donor program.

Our approach for automated software transplantation extracts the organ from the donor and implants it into the host. But the organ itself is a library with a well defined interface. The type signature of the organ is defined by the implantation point into the host: the formal arguments of the *organ* (that is a function) are all the variables available in the host at the implantation point.

It would be a trivial extension to μSCALPEL to allow an user to specify a desired type signature for the organ, rather than inferring it from the host at the implantation point. This extension of

μSCALPEL would then not require a host any more if we want to extract a functionality from a donor program into a library. The inputs in this case would be: the organ entry point, the formal arguments of the desired functionality, and the organ test suite. Under this, μSCALPEL would be capable to extract libraries for programs that can be reused across multiple programs (or hosts as we call them in autotransplantation terms).

Software clones [166] refer to duplicated passages of source code that appear in the same software system. Although our organs can be seen as clones of the code in the donor program, they are used in a different program (*i.e.* the host), making them more similar to libraries than to clones.

An alternative to software transplantation could be using directly the donor program to execute the organ's functionality. This might be feasible if the most of the donor system is the organ but, in general, the organ is only a small part of the donor system. Additionally, the donor system might not be a CLI. In this case, reaching the organ's functionality in the donor might take a while in the donor's user interface.

The organ might use the inputs in a different way than they are available in the host system. μSCALPEL automatically does the transformation of the data in the host to the ones expected by the organ with the organ's vein. To use the organ from the donor one would have to execute the host until the point where the inputs for the desired functionality are available, transform them for the usage in the donor, and finally execute the donor until the point where he/she can add the transformed inputs in the donor. This would highly affect the usability and the user experience with the organ's functionality. Autotransplantation provides a native experience of the organ in the host program.

4.6 Research Questions

This section reports the research question which we aim to answer, in the autotransplantation work. We illustrate our current answers to this questions through the empirical study(see Section 4.7, and Section 4.8), and the 2 case studies (see Section 4.9, and Section 4.11).

Since our approach transplants code from the donor into the host, a natural first question to ask is "Does transplantation break anything in the host program?". In biological terms, we are checking the side-effects of the transplantation. In software engineering, this question becomes one of regression testing: "Does the modified host pass all the regression test cases?". If it does, then we conclude that we have found no evidence for any side effects of the transplant operation. This motivates our first research question:

Research Question 1: Can we transplant a software organ into host that, after transplantation, still passes all of its regression tests?

Of course, the answer to this question depends critically on the quality of the regression testing. All the programs with which we experimented are real-world systems equipped with test suites, deemed to be useful and practical by their developers. However, they were not designed to test transplants and may not be sufficiently rigorous to find regression faults introduced by transplantation.

We computed coverage information for each subject's test suites and found that it was not always high. Finding test suites that achieve high coverage of real-world code using system-level testing is a known challenge. Despite much work on tool development, automating extensive coverage in system-level testing remains an open problem [174]. Furthermore, the value of higher coverage in testing is the subject of ongoing debate in the research community [141].

Nevertheless, since we are injecting new code into the host from a foreign donor, we certainly ought to seek to cover the host system as thoroughly as possible in our testing. We therefore manually augmented the each subject's existing regression test suites with additional test cases to increase coverage. Our aim is to more rigorously regression test our transplant operations for side effects. This motivates our two specific versions of RQ1, which focus on each of these coerces of regression test suite:

RQ1.1: Can we transplant a software organ and still pass all of the postoperative host's existing regression tests?

RQ1.2: Can we transplant a software organ and still pass a regression test suite, manually augmented to achieve high coverage in the postoperative host?

Achieving transplantation without side effects is necessary, but not sufficient for success. We need to do more than merely insert alien code into the host without breaking regressions tests; we need to augment the functionality of the postoperative host with new behaviour that replicate the software organ we extracted from the donor, or the operation is pointless. This motivates our second research question:

Research Question 2: Does the transplanted organ provide any new functionality to its postoperative host?

We consider two approaches to test whether the postoperative host exhibits the transplanted functionality. The first, natural question to ask is whether the postoperative host passes acceptance tests. That is, system-level tests that specifically target the new, hoped for, behaviour that would represent an implementation of the new feature in the host. For this purpose, we need to adapt the donor's acceptance test suite to be suitable for the postoperative host, *i.e.* define τ in Definition 1. In the most cases, the donor's inputs and the host's are very different. Thus, we have manually defined the τ function, for each transplantation. For example, Cflow takes a set of C source code files as input, while Pidgin is a GUI program. Thus, to supply Cflow with its inputs, after its insertion into Pidgin, we added code to Pidgin that opens a dialog to prompt the user for files. To validate the transplantation, we then compared the output of the Cflow organ in Pidgin against the original Cflow over same set of files.

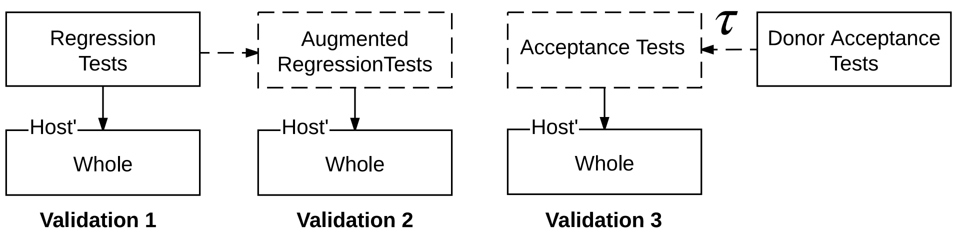


Figure 4.14: Three validation steps; the dashed boxes are test cases we created.

In summary, our first two research questions draw on three different forms of validation, as outlined in Figure 4.14. If we can find transplants that pass regression tests and transfer a meaningful amount of new functionality into the host, satisfying all three of these validates steps, then this would be encouraging evidence that automated code transplantation is a feasible means of extracting and transplanting functionality from donors to hosts. However, there remains the question of the computational cost of this overall approach:

Research Question 3: What is the computation effort to find these organs?

The first three questions are answered entirely quantitatively. With these questions, we give initial empirical evidence to support our claim that μTrans to automated reuse merits further consideration.

However, even if the answers to all of these questions are generally positive and encouraging, there remains the question of whether automated code transplantation can deliver *useful* new functionality. This motivates out final research question:

Research Question 4: Can automated transplantation transfer useful, new functionality into the host system?

To answer this question we use μSCALPEL to automatically transplant: 1) a feature into a host that was also developed by human developers; and 2) 2 features that were requested on the development forum of the host system. This allows us to qualitatively and quantitatively study an instance of transplantation in which we *know* the goal of transplantation is useful, since humans sought to achieve the same results using traditional manually intensive programming (in the first case), or have requested them for the host system (in the second case). We use 2 case studies for answering to this research question.

4.7 Empirical Study

This section explains the subjects, test suites, and research questions we address in our empirical evaluation of automated code transplantation as realized in our tool, μSCALPEL.

Subjects We transplant code for five donors into three hosts. We used the following criteria to choose these programs. First, they had to be written in C, because μSCALPEL currently operates only on C programs. Second, they had to be *popular* real-world programs people use. Third, they had

Table 4.1: Donor and host corpus for the evaluation.

			Tests	
Subjects	Type	Size	#Regr.	#Unit
Idct	Donor	2.3k	-	3-5
Mytar	Donor	0.4k	-	4
Cflow	Donor	25k	-	6-20*
Webserver	Donor	1.7k	-	3
TuxCrypt	Donor	2.7k	-	4-5
Pidgin	Host	363k	88	-
Cflow	Host	25k	21	-
SOX	Host	43k	157	-
Case Study:				
x264	Donor	63k	-	-
VLC	Host	422k	-	-

to be *diverse*. Fourth, the host is the system we seek to augment, so it had to be large and complex to present a significant transplantation challenge, while, fifth, the organ we transplant could come from anywhere, so donors had to reflect a wide range of sizes. To meet these constraints, we perused GitHub, SourceForge, and GNU Savannah in August 2014, restricting our attention to popular C projects in different application domains.

Presented in Table 4.1, our donors include the audio streaming client IDCT, the simple archive utility MYTAR, GNU Cflow (which extracts call graphs from C source code), Webserver[8] (which handles HTTP requests over both IPv4 and IPv6), the command line encryption utility TuxCrypt, and the H.264 codec x264. Our hosts include Pidgin, GNU Cflow (which we use as both donor and host), SOX, a cross-platform command line utility that can convert and apply effects to to audio formats, and VLC, a media player. We use x264 and VLC in our case study in Section 4.9; we use the rest in Section 4.8.

These programs are diverse: their application domains span chat, static analysis, sound processing, audio streaming, archiving, encryption, and a web server. The donors vary in size from 0.4–63k SLoC and the hosts are large, all greater than 20k SLoC. They are popular: Wikipedia reports that more than 3 million people used the Pidgin chat client in 2007. Pidgin, SOX, and TuxCrypt are downloaded from sourceforge over a million times each year on average. VLC, MyTar, and webserver average 105 forks and 148 watchers on GitHub. x264 is an award winning and GNU Cflow is a well-known, well-maintained static analysis tool.

Test Suites We use three different test suites to evaluate the degree to which a transplantation was successful: 1) the host's pre-existing *regression* test suite, 2) a manually augmented version of the host's regression test suite (*regression*++), and 3) an *acceptance* test suite for the postoperative host, manually updated to test the transplanted functionality at the system-level. A host's developers

[8]https://github.com/Hepia/webserver.

designed its pre-existing regression test suite and distributed it. Sadly, these suites do not always achieve high statement coverage. We use these test suites to answer **RQ1.1**. To achieve higher statement coverage, we manually added tests to a host's pre-existing test suite to create regression++, our augmented regression suites, for each host. We use these test suites to answer **RQ1.2**. Finally, we manually realized Definition 1, defining input adaption τ and equivalence $\simeq$ for the test oracle, to create an acceptance suite for the postoperative host: starting from the donor's acceptance tests, one of the authors devised acceptance tests for the postoperative host that execute the transplanted software organ from the postoperative host's entry point. They assess whether the new functionality required is to be found in the host. We use these to answer **RQ2**. Our corpus and our test suites are available at `http://crest.cs.ucl.ac.uk/autotransplantation`.

To thoroughly validate the results of the transplants that μSCALPEL did, we kept these three test suites completely hidden from μSCALPEL. In the GP process, μSCALPEL uses the organ test suite, while for the evaluation of the transplants we used three different test suites to be sure that the evaluation is independent from the implementation of μSCALPEL and to be sure that our transplants do not overfit the test suite that is used in the GP process. That is, the organ test suite is used only by μSCALPEL in the GP process, while the evaluation test suites are used just for the evaluation of the transplants and are invisible for μSCALPEL.

4.8 Results and Discussion: Empirical Study

For all 15 experiments, we report the number of runs in which all test cases passed in all test suites. We call these *unanimously passing runs* and report them in Table 4.2. We also report the number of successful runs for the regression, augmented regression and acceptance test suites (columns *PR* in Table 4.2). We repeat each experiment 20 times. Before we answer the research questions posed, we summarise our results.

We transplanted an archiving feature from MyTar. Generation of inverse discrete cosine transform coefficients from an array of integers was extracted from the IDCT donor, while the AES encryption feature was extracted from TuxCrypt. We also transplanted our Web donor's functionality that starts a web server and provides file access. Parsing and processing C source code was the feature extracted from Cflow.

Table 4.2 provides evidence that automatic software transplantation is indeed feasible. *In 12 out of 15 experiments all test cases passed*, while 63% (188/300) of all the runs unanimously passed all test suites. *New functionality from 4 out of 5 different donors has been successfully transplanted into the three chosen host systems.* Moreover, within 20 repetitions we were successful in at least 13 runs in these cases. Given that we set ourselves the challenging task of transplanting new functionality into an existing system, μSCALPEL's success rate of at least 65% shows the great promise of μTrans.

Furthermore, our regression failure in the case of the Web donor is that the automatically extracted code contains a loop that listens for server requests. Regression tests were designed for the

Table 4.2: Transplantation results over 20 repetitions: column *unanimously passing runs* shows the number of runs that passed all test cases in all test suites; column *Test Suites* shows the results for each test suite: *PR* reports the number of passing runs; *All* and *O* report statement coverage (%) for the postoperative host and for the organ; column *Time* shows the execution time (User+Sys) in minutes; * excludes tests that failed due to a pre-existing bug in the organ.

		Unanimously	Test Suites									Time (min)	
Donor	Host	**Passing**	Regression			Regression++			Acceptance			Avg.	Std.
		Runs (PR)	PR	All	O	PR	All	O	PR	All	O		Dev.
IDCT	Pidgin	**16***	20	8	0	17	51	99	16*	40	99	5	7
MyTar	Pidgin	**16**	20	8	0	18	51	93	20	40	61	3	1
Web	Pidgin	**0**	20	8	0	0	51	65	18	40	65	8	5
Cflow	Pidgin	**15**	20	8	0	15	52	53	16	41	54	58	16
TuxCrypt	Pidgin	**15**	20	8	0	17	51	88	16	40	88	29	10
IDCT	Cflow	**16**	17	48	91	16	69	91	16	50	99	3	5
MyTar	Cflow	**17**	17	50	52	17	69	90	20	50	91	3	¡1
Web	Cflow	**0**	0	3	13	0	61	68	17	49	62	5	2
Cflow	Cflow	**20**	20	71	70	20	71	70	20	71	70	44	9
TuxCrypt	Cflow	**14**	15	46	66	14	69	87	16	50	83	31	11
IDCT	SOX	**15**	18	32	94	17	42	94	16	20	94	12	17
MyTar	SOX	**17**	17	31	62	17	42	90	20	22	60	3	¡1
Web	SOX	**0**	0	12	27	0	12	67	17	12	65	7	3
Cflow	SOX	**14**	16	29	47	15	41	66	14	19	59	89	53
TuxCrypt	SOX	**13**	13	35	79	13	41	86	14	20	79	34	13

host prior to transplantation and may not handle nonterminating behaviour. However, in at least 17 runs, the postoperative host passed its acceptance test suites. The Web donor's organ is reachable in the postoperative host and its behaviour retained in all three hosts. When transplanting functionality from Cflow into itself we achieved 100% success rate. This confirms our suspicion that this case would be the easiest for our approach, since issues such as implantation point, variable scope and renaming become less of a concern.

Once we extracted a new functionality, we insert it into the host programs. We ran the test suites provided with Pidgin, Cflow and SOX to check if these software systems retained features deemed important by software developers.

RQ1.1: *Can we transplant a software organ and still pass all given regression tests?* In all but two cases, the beneficiaries pass the regression suites in at least 13 of 20 runs (Table 4.2). Web servers are reactive systems, which contain an event loop. The webserver donor's organ contains such a loop, in which it listens for http requests. Some of Cflow and SOX tests enter the event loop, do not terminate, and fail. *Therefore, the answer to RQ1.1 is that new functionality has been inserted in 13 out of 15 experiments without distorting host program behaviour exercised by the given test suites.* In all transplantation experiments, Pidgin, on the other hand, passed all regression tests. Given this variance, we use `gcov`, a popular coverage tool, to measure the path coverage of our subject's test suites. Table 4.2 reports our results. The *All* columns report coverage rates for the entire host program with new functionality transplanted, while the *O* columns report statement coverage just for the organ. For Pidgin, the statement coverage rate is only 8%, while the transplanted code is not covered at all.

RQ1.2: *Can we transplant a software organ and still pass a regression test suite, manually augmented to achieve high coverage in the postoperative host?* We augmented the hosts' regression test suites with additional tests to increase statement coverage, which we report in Table 4.2. The new test suites now cover 41–71% of the host programs with transplanted features, with one exception, while organ coverage is even higher, exceeding 90% in 6 cases. Furthermore, the success rate was largely retained for Cflow and SOX, where we lost at most one successful passing run for the augmented regression test suite vs. the pre-existing one, as shown in Table 4.2. Coverage rates of Pidgin increased from 8% to 51–52% which had an impact on success rates. The augmented test suite covered 53–99% of the organ's code. Since the organ from Web donor was not covered by the original regression test suite and the organ contains an infinite loop, as pointed out above, the augmented suite has a pass rate of 0%. *Therefore, our answer to RQ1.2 is that postoperative hosts passed all of their augmented regression tests in 12 out of 15 experiments.*

RQ2: *Does the transplanted organ provide any new functionality to its postoperative host?* Next, we test whether the organ is reachable within the host and if it retains the desired behaviour. The numbers of successful runs of these acceptance test suites, and the statement coverage are presented in Table 4.2. The statement coverage ranges from 54 to 99% for the organs and 12 to 71% for the whole host program with transplanted new functionality. Since these tests are aimed at the organ code, we did not expect to get high coverage in the hosts, which is the lowest, 12–22%, in the case of SOX. We suspect that dead code, from older audio formats that are no longer supported, is the reason. However, organ coverage always exceeds 59%. The highest coverage was achieved for Cflow, which contains recursive calls.

The number of times the entire acceptance suite passed ranges from 14 to 20. This means that new features are indeed found in the hosts in at least 70% of the repeated runs. Moreover, all subjects with new functionality from MyTar pass all acceptance tests in all 20 runs, hence the organ is always reachable. Furthermore, by transplanting a piece of IDCT into Pidgin, we discovered a bug in the original code, not reachable in the donor. With Pidgin as its host, however, the IDCT organ can be reached with an empty array which the code does not handle. The success rates presented in Table 4.2 are based on the number of tests passed excluding our test cases that discover the bug, since it is present in the donor and we measure faithfulness to the original program. Therefore, not only we have shown that we can transplant code from one program into another, but also these new features will be reachable and executable. *To answer RQ2, in all 15 experiments the transplanted features are accessible and provide the desired functionality. Moreover, 85% (256/300) of the runs passed all acceptance tests.*

RQ3: *What is the computation effort to find these organs?* We have shown that we can successfully transplant new functionality into an existing software system. The question remains, how efficient is our approach? Average timings of extracting and transplanting a new feature into the host program are shown in Table 4.2 (column *Avg*). *To answer RQ3, average runtime of one run of the transplantation process did not exceed 1.5 hour and in 8 cases it was less than 10 minutes.* This shows the efficiency of our approach. If you consider the time it would have required for a human to perform the same tasks of adding new functionality, we believe that these timings would have been in the order of hours or days rather than minutes.

4.9 Case Study: Transplanting H264 from x264 into VLC

Despite the fact that the answers to research questions 1–3 in Section 4.7 are generally positive and encouraging, the question of whether automated code transplantation can deliver *useful* new functionality remains: **RQ4:** *Can automated transplantation transfer useful, new functionality into the host system?* We answer this question by considering the automated transplantation of a feature into a host that human developers implemented. This allows us to qualitatively and quantitatively study the transplantation of a feature whose transplantation we *know* to be useful, since humans worked to achieve the same results using traditional, manually intensive programming.

VLC is a popular open source media player[9]; VLC seeks to offer a versatile media player that "can play any file you can find". To achieve this goal, VLC is under continual development, especially to handle new media formats. x264 is a popular, widely-used, award winning, open source video and audio encoding tool[10], designed for the H.264/MPEG-4 AVC compression format[11]. Both programs are substantial: at the time of this writing, x264 contains more than 63k lines of code and 211 files; version 2.1.5 of VLC contains more than 422k lines of code across 2821 files.

H.264 is an evolving video standard, to which features are continually added. Keeping a code base current with an evolving standard like H.264 is time-consuming. Currently the developers of VLC manually update the code related to x264 library, when its interface changes. VLC has invested considerable effort in supporting H.264, first implementing it in November 2003, before it was standardised. Since then, over 450 commits in VLC's version history mention H.264. These commits run the gamut, including bug fixes, new features, or refactoring. The most recent change to mention H.264 was committed 21 January 2015. VLC's developers have worked over more than 11 years at maintaining and updating the handling of the H.264 format.

To insulate themselves from changes in x264, the VLC community includes it as a library veiled by wrapper, which they must update when it changes. Autotransplantation can sped this process by automatically handling renaming and, as we describe below, it also takes only that part of the x264 codebase that it actually uses. To use μSCALPEL here, a VLC developer strips out the all the code at an implantation point (conceptually, this removes the previous version) as μSCALPEL implants an organ as an atomic, it does not merge it into existing code.

[9]http://www.videolan.org/developers/x264.html.
[10]http://www.videolan.org/vlc/.
[11]http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC.

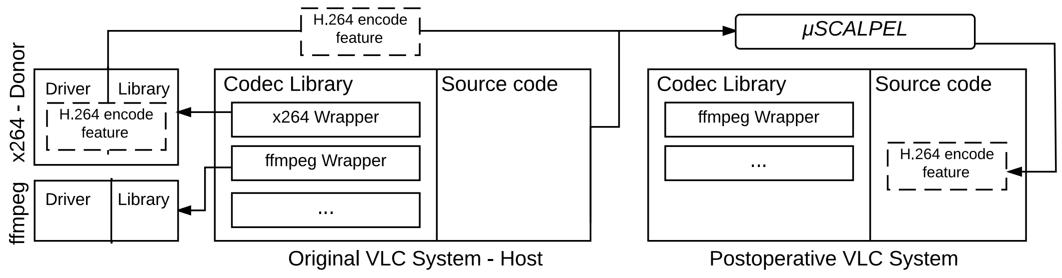


Figure 4.15: Autotransplanting x264's encoder functionality into VLC; the donor is the source code of x264, the organ its H.264 encoder not the entire library, and the host is VLC, stripped of any reference to x264 prior to transplantation; µSCALPEL transplants the encoder functionality as distinct from x264's incarnation as a library.

Consider Figure 4.15. At the top left is the original VLC system, which links all of x264 as external library and interacts with its encoder through a wrapper. At the bottom left is the host, which we produced from original VLC codebase by stripping all references to x264, by building it using `autoconfig --enable-x264=no`; this setting does not link the x264 library and eliminates the wrapper via preprocessor directives. This step ensures that µSCALPEL transplanted new, rather than uncovering old, functionality.

We marked `x264_encoder_close` (`encoder.c` : 2815) as the organ entry point and defined an organ test suite by using the original x264 implementation as the oracle. In the host, we annotated `input_DecoderNew` (`decoder.c` : 314) as the implantation point, because VLC calls this method for encoding or streaming a video, and choosing what encoder to use.

Using the organ entry point annotation and the organ test suite, µSCALPEL extracts the encoder organ, the box labeled "H.264 encode feature" in Figure 4.15, from the donor, then implants it into the stripped version of VLC at the designated implantation point to produce a postoperative version that includes x264 encoder functionality. Due to its use of slicing, µSCALPEL takes only what VLC needs and uses from x264, unlike the human process which replaces the x264 as a library and updates its wrapper.

4.9.1 Study Design and Setup

x264 is multi-threaded, and so is the code organ that µSCALPEL extracted from it and transplanted. We transplanted the H.264 organ from x264 with the assembly optimisations disabled, because µSCALPEL is built on TXL's C grammar, which does not handle assembly code. The latest version of VLC (2.1.5) was our target host, after manually disabling its existing H.264 functionality by building it with `autoconfig --enable-x264=no`. Under this setting, the x264 library is not linked and the preprocessor eliminates VLC's x264 wrapper and leaves VLC without the capabilities of encoding H.264 videos. Thus, VLC is not any more set up for the execution of the x264 code. As noted above, the insertion point is `input_DecoderNew`, VLC's encoder selection method.

Id	Video	Length (s)	Frames	Bit rate
1	The Tale of Princess Kaguya	10	150	3218
2	The Tale of Princess Kaguya (full)	59	886	6525
3	Jersey Shore Massacre	46	693	8436
4	The Pyramid	138	2072	8680
5	Teenage Mutant Ninja Turtles	150	2260	9419

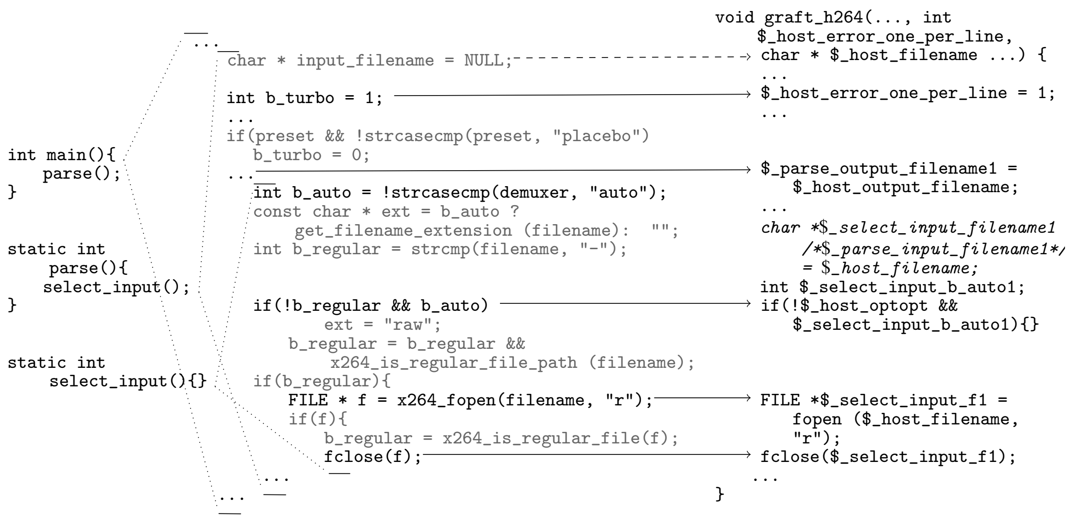
Figure 4.16: The video clips used in our case study.

Before we could apply μSCALPEL to x264, we had to manually workaround TXL's limitations in handling C preprocessor directives: by default, TXL treats preprocessor directives as strings. Pragmatically resolving an `#ifdef` as a string requires replacing it with its `then` branch or its `else` branch or both. In any nontrivial program, like x264, any of these choices either fails to compile or generates invalid code. Note that we only process with TXL the donor program (*i.e.* x264). The code of VLC is not touched by our TXL tools and thus, in the case of the elimination of the *x264* wrapper with `autoconfig --enable-x264=no` the preprocessor behaves in the same way as in the initial VLC program and correctly eliminates VLC's x264 wrapper. The only modification that automated software transplantation does in the case of the host program's source code (*i.e.* VLC) is to add a call to the organ entry at the implantation point.

For our evaluation, we chose 5 movie trailers, with lengths varying between 10 and 138 seconds, frames between 150 and 2260, and bit rates between 3218 and 9419. Figure 4.16 details the 5 movie trailers. We ran both of our versions of VLC — original and with the transplant — in command line mode, bypassing its GUI. We used VLC's command line parameter `vlc://quit` to close VLC when a video ends. We used the same encoding options across all videos and all program versions: x264 without and with x264 ASM optimisations, original VLC, and VLC with the x264 organ implanted. These options are "4:4:4 Predictive" for the profile option (colour handling), 4:2:0 for YUV (luminance and chrominance weight), and 2.2 for level (maximum bit rate).

4.9.2 Observations

We transplanted a specific version of x264 into VLC. We cannot precisely measure how much time the VLC community spent updating their code for new versions of x264, so we bound it using VLC's version history. Over 11 years, the VLC community has upgraded VLC many times. We assumed that these upgrades triggered a burst of commits, where we consider a burst to be sets of four commits no more than 10 days apart. In VLC's version history, we identified a total of 39 commit bursts whose logs contain '264'. On average, an H.264 commit occurred every 9 days. The average number of commits across the 39 bursts is 8. 312 commits were in bursts; while 126 were isolated. Each commit burst spanned 20 days on average. We uniformly selected the commit burst starting 5 January 2012. and ending 21 January 2012. It has 12 commits. Its commits changed 2 files. The `diffstat` of these commits is 171.



```
int main(){
    parse();
}

static int
    parse(){
    select_input();
}

static int
    select_input(){}
```

```
...
char * input_filename = NULL;

int b_turbo = 1;
...
if(preset && !strcasecmp(preset, "placebo")
   b_turbo = 0;
...
   int b_auto = !strcasecmp(demuxer, "auto");
   const char * ext = b_auto ?
        get_filename_extension (filename):  "";
   int b_regular = strcmp(filename, "-");


   if(!b_regular && b_auto)
         ext = "raw";
      b_regular = b_regular &&
           x264_is_regular_file_path (filename);
   if(b_regular){
      FILE * f = x264_fopen(filename, "r");
      if(f){
         b_regular = x264_is_regular_file(f);
         fclose(f);
   ...
...
```

```
void graft_h264(..., int
     $_host_error_one_per_line,
     char * $_host_filename ...) {
     ...
     $_host_error_one_per_line = 1;
     ...

     $_parse_output_filename1 =
         $_host_output_filename;
     ...
     char *$_select_input_filename1
         /*$_parse_input_filename1*/
         = $_host_filename;
     int $_select_input_b_auto1;
     if(!$_host_optopt &&
         $_select_input_b_auto1){}

     FILE *$_select_input_f1 =
         fopen ($_host_filename,
         "r");
     fclose($_select_input_f1);
    ...
}
```

(a) x264 Original Code (b) x264 Inlined Source Code (c) Transplanted Source Code

Figure 4.17: Transplant operation in the case study. Code snippet from the beginning of the graft. ⋯| means function inlining; `input_filename` is mapped to `$_host_filename`; ⟶ means original statement replacement under α — renaming; grayed statements are deleted. The conditional `if(f)` was dropped from the organ, because reaching the organ in the host was not possible with an empty file.

The total size of the organ μSCALPEL extracted was 23k LOCs, including the veins, as reported by `cloc`. In contrast, an over-approximate estimate of the equivalent human organ is 12k in VLC and 44k SLoC in its x264 library. Half of the μSCALPEL's organ's lines are global variable declarations; type and function definitions consume half the lines; inlined function calls are the next largest source of lines. μSCALPEL performs standard renaming, but is also able to reuse declarations in the host. For example, both VLC and x264 contain the typedef `uint8_t`. In the vein, we have `uint8_t output_csp_fix[]`; in the host, we have `uint8_t *p` in scope. Since the types are compatible, μSCALPEL tries to unify them. If this succeeds, μSCALPEL removes `uint8_t` from the organ, since it is no longer needed. The organ likely contains unnecessary functions. In the future, one could post-process μSCALPEL's output to shrink the organ, as proposed in genetic improvement work [178, 248].

Figure 4.17 shows μSCALPEL in action. On the left, Figure 4.17.a contains a snippet of code as it appears in the donor, after slicing has identified it as part of the vein in the donor. Figure 4.17.b shows the organ after extraction into the ice-box, with all the functions in the vein inlined. The dotted lines bracket the regions inlined from Figure 4.17.a to Figure 4.17.b. At the right, Figure 4.17.c shows the organ after transplantation into the host. From Figure 4.17.b to Figure 4.17.c, the dashed lines show deletions and the solid lines show variable equivalences. At the right, we see `graft_h264`, the entry point into the organ in the host. Its parameter list is all variables in scope at the insertion point in the host. The organ may not use all of these parameters. At entry, `graft_h264` hooks the host variable (as parameters) to the organ's variables, then executes the code from the organ's vein to initialize state before calling the organ. This is why this snippet has many deletiong and is dominated by assignment

```
I   <<h = x264_encoder_open_142 (param);>>
+   x264_param_t *$_x264_encoder_open_142_param1 = $_encode_param1;
+   x264_t *$ABSTRETVAL_ret_x264_encoder_open_1421;
α   x264_t *$_x264_encoder_open_142_h1;
    do {
        do {
I         <<x264_malloc (sizeof (x264_t))>>
+         $_host_error_one_per_line = sizeof (x264_t);
/         uint8_t *$_x264_malloc_align_buf2;
/         $_x264_malloc_align_buf2 = ((void *) 0);
-         if (i_size >= 2 * 1024 * 1024 * 7 / 8) {
α           $_x264_malloc_align_buf2 = memalign (2 *1024 * 1024,$_host_error_one_per_line);
-           if (align_buf) {
α             size_t $_x264_malloc_madv_size2 = ($_host_error_one_per_line + ...);
α             madvise ($_x264_malloc_align_buf2, $_x264_malloc_madv_size2, 14); }
-           else align_buf = memalign (32, i_size);
    ...
-     return $_x264_encoder_open_142_h1;
+     $ABSTRETVAL_ret_x264_encoder_open_1421 = $_x264_encoder_open_142_h1;
+     goto LABEL_x264_encoder_open_142_1;
    ...
-   return ((void *) 0);
+   $ABSTRETVAL_ret_x264_encoder_open_1421 = ((void *) 0);
+   LABEL_x264_encoder_open_142_1:
+   $_encode_h1 = $ABSTRETVAL_ret_x264_encoder_open_1421;
```

Figure 4.18: Diff of code transplanted from the 'library' part of x264. 'I' means inlining, '+' means line addition, 'α' means α–renaming, while '/' means initialisation separation. All the lines with '-' are in the namespace of the original Donor. μSCALPEL has specialised the function 'x264_malloc', for the current calling context. Here, 'x264_malloc' is called for allocating memory for 'x264_t' data type. Its size is always greater than the value checked in the first 'if', so it was deleted.

statements. For example, the organ reads the video stream from `$_host_filename`. We use TXL's built facility for generating fresh names to avoid variable capture to generate the names you see. As you can see, μSCALPEL reduces an if statement to a NOP; organ mininization is future work.

Figure 4.17 focuses on initialization code; Figure 4.18 shows how μSCALPEL modified functional organ code during autotransplantation. x264 heavily uses `x264_malloc`, its wrapper for the standard `malloc` function. μSCALPEL specialises it for different calling contexts. Here, x264 has called it to allocate memory for `x264_t`, which is bigger than `2 * 1024 * 1024 * 7 / 8`. This allocation always succeeded in our test cases, so μSCALPEL eliminated the unneeded if statements, as shown. Since function calls can occur in arbitrary expression and μSCALPEL greedily inlines their definition upon encountering them, it replaces return statements with assignment to a fresh variable. To skip any code that follows the replaced return, whose execution the return blocked, μSCALPEL follows the assignment with a GOTO to a label after the call in the caller (2 in Figure 13). When a declaration follows the added LABEL, as with void functions, μSCALPEL generates a NOP to preserve semantics correctness, without affecting semantics.

μSCALPEL needed only a single run and 26 hours to extract and transplant x264's H.264 encoder into VLC; in other words, it passed all the regression, augmented regression and acceptance test suites. Since statement coverage for VLC's existing test suite was 11.5% and 0% for the organ, we manually added tests. The resulting, augmented test suite combined with our acceptance tests achieved 63%

coverage of the beneficiary and 49.4% of the organ. After 50 runs over our test cases (10 for each video trailer), gcov reported a coverage of 49.4%.

As mentioned, μSCALPEL works on C, not assembly language. In the case of video encoding, ASM optimisations vastly improve encoding speed. Because of this limitation, our runtime results are up to 8 times slower than those of the original VLC program. Here again, we can, in the future, turn to GP to optimise efficiency of the beneficiary: a recent technique speed up existing code 70-fold [178, 248]. We also measured the size of the encoded video. The output of our transplanted organ matches the size of its output when run in the donor, in every case. However, the size of the output of our organ after transplantation is up to 1.7 times as big as in the original VLC. This is because, we provided μSCALPEL with test cases that tested x264's lossless encoding, which μSCALPEL therefore extracted into the organ, not x264's lossy encoding; VLC, in contrast, uses lossy encoding. In general, the x264 organ in VLC cannot be configured any more in this transplant: the x264 configuration settings became hardcoded to the ones that our test cases specify. In principle it would be possible to keep the x264 organ configurable, by providing test cases that check different configurations. But this would have required an additional component in VLC to be able to select the right configuration settings. Such component could have been transplanted in a different automated software transplant from the same x264 donor. Combining multiple transplants to work together is future work for automated software transplantation.

To answer RQ4, our tool needed just 26 hours for extracting and transplanting the H.264 multi-threaded encoding feature from x264 into VLC.

4.10 Humies Gold Medal

In this section we present our submission to Humies. We were awarded the gold medal for human competitive result, together with a prize of 5000$[12].

Humies[13] is a competition part of the Genetic and Evolutionary Computation (GECCO) conference [14] that provides awards totaling $10,000 for human competitive results that an evolutionary computation approach produced. We entered with our H.264 transplant from x264 donor program into the VLC media player (Section 4.9 details this case study) into Humies 2016. We are grateful to the judges from Humies 2016 that awarded us the gold medal together with a cash prize of $5,000. This chapter reports our Humies entry.

Humies Criterion: an entry to Humies is considered to be *human-competitive* if it satisfies at least one out of eight different criteria, listed on Humies website[15]. We showed that our automated software transplantation of H.264 encoding functionality satisfies five *human-competitiveness* criteria:

[12]http://gecco-2016.sigevo.org/index.html/Humies.html
[13]http://www.human-competitive.org
[14]https://en.wikipedia.org/wiki/Genetic_and_Evolutionary_Computation_Conference
[15]http://www.human-competitive.org

- **The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts:** H.264 is the most widely used media encoding format, because of its increased compression rate, speed, and quality. There are a lot of media encoders that encode in H.264 format. These encoders are regularly compared against each other, as numerous online encoder databases and archives show (http://www.compression.ru/video/codec_comparison/h264_2012/ ; http://web.archive.org/web/20110719095845/ http://www.tkn.tu-berlin.de/research/evalvid/EvalVid/vp8_versus_x264.html). We transplanted the H.264 encoding functionality from x264, the tool recognised as the best H.264 encoding by these databases and archives. Our transplanted "organ" performs slightly better than the original implementation of H.264 encoder in the x264 donor program.

- **The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions:** Media encoding is a long-standing problem in computer science. We have seen an continually increasing boost in the video quality. This increase cause a problem in terms of disk space requirements. Even if the disk space is pretty cheap now, a high quality movie still represents a problem from this point of view. Now, 1920x1080 quality is standard, although 4k videos are starting to be more used. Just an 1 hour movie at 1920x1080 resolution, and 60 FPS takes on the raw stream (not encoded version) 1.53 TB hard disk space (http://studiopost.com/contact/tech-specs/calculating-disk-space-requirements?format=uncompressed_10_1080&frame_rate=f60&length=1&length_type=hours). The most of users are not having their entire HDD capacity not even near to this requirement. On the contrast, a H.264 encoded version of the same video, require only 86.9 GB (http://studiopost.com/contact/tech-specs/calculating-disk-space-requirements?format=h264_1080&frame_rate=f60&length=1&length_type=hours), so just 0.05% of the initial space requirement. For solving these problems, a lot of encoding standards are created. Out of this, H.264 is the most recent human-created solution. We slightly outperform the best H.264 (as the results to various competitions show) implementation, from x264.

- **The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered:** H.264 encoding was a big improvement over previous media encoders. x264 is the award winning implementation of H.264. Our experiments show that the transplanted H.264 encoder, from x264, performs slightly better than the x264 donor program.

- **The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs):** There are a lot of H.264 regulated competitions that evaluate various human-written H.264 encodings. x264 won the following competitions: first place at MSU Sixth MPEG-4 AVC/H.264 Video Codecs Comparison, with 24% better encoding than second place; first place at Doom9's 2005 codec shoot-out, passing Ateme by a hair; tied for 1st place (with Ateme) in the second annual MSU MPEG-4 AVC/ H.264 codecs comparison. (more details about these competitions on `http://www.videolan.org/developers/x264.html`). Our transplanted H.264 organ from x264, into VLC, performs slightly better in terms of encoding time, than the same version of the x264 program.

- **The result solves a problem of indisputable difficulty in its field:** With the abundance of code available online software developers need not write their programs from scratch. If they require a particular feature, for example a video encoding functionality for their media player, they can choose to use an open source implementation and transplant it into their program. However, this is a non-trivial task that has thus far been performed manually. We present an automated approach and show several successful autotransplantation results. In particular, in 26 hours of computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the popular open source VLC media player. We estimate that upgrade of x264 within VLC took human programmers an average of 20 days of elapsed, as opposed to dedicated, time (based on VLC's git version history).

Genetic programming, and genetic improvement, were successfully applied for fixing a lot of real world problems: improving execution time, bug fixing, improving energy consumption, etc. GP, and GI, usually consider non-functional properties of an existing program. In Humies 2016 submission, we automatically transplanted the best implementation of H.264 encoder standard, into VLC, the most used open source media player. Both systems are non trivial, popular, and substantial real world systems.

Implementing video encoders is a tedious, error prone, and difficult activity, for any video player. Any modern video player should handle H.264 encoding, since it is the most widely used. Implementing this standard from scratch is very difficult, and requires specialised multimedia, and mathematical knowledge. Also H.264 is a continually evolving standard, with a total of 22 versions up to now. Wouldn't be great if we were able to take the best existing implementation of the latest H.264 standard, and automatically move it into our own video player? What if we would be able to even obtain a slightly better version of that implementation? This is what we did in our H.264 organ transplantation experiment (Section 4.9), with very promising results. We are the first to use GP/GI to automatically move useful functionality, between two potentially unrelated systems. We know the functionality is useful, since the developers of VLC are maintaining the H.264 encoders, since 14 years ago. We know this is not a trivial task, from the GIT History of VLC. We identified a total of 438 commits about H.264 standard.

We are the first one to use code transplants for moving non-trivial, and useful functionality between two unrelated systems. We implemented our approach in μSCALPEL, and used it to transplant the most used media encoder (H264), from an award winning donor program (x264), into a very popular open source video player (VLC). As a side effect of our transplantation process, we slightly improved the initial encoding speed of the H264 ”organ” from x264.

4.11 Case Study: Automatic Transplantation of Call Graph and Layout Feature into Kate

In this section, we illustrate the way in which realistic, scalable, and useful real-world transplantation can be achieved using μSCALPEL, in a case study. We apply our tool to the SSBSE 2015 Challenge program `Kate`[16], a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. We perform two automated transplantations using μSCALPEL. In the first one, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs.

4.11.1 Applying Autotransplantation to Kate

We chose two popular, real world systems, as the donor programs: GNU cflow[17], and GNU `indent`[18]. The former generates call graphs for C programs, a feature is currently missing from Kate; the latter provides a pretty prints C source files, with far fewer restrictions than Kate’s existing built-in indentation functionality. Kate’s existing indentation feature fails to wrap a line that is too long, for example. It simply adds space or tabs in a programming language independent manner, whereas GNU’s `indent` exploits language awareness to provide far better formatting functionality.

μSCALPEL requires user to provide an implantation point in H, and the entry point of the feature in D. We chose one of the Kate plugins as the implantation point. We start from the time date plugin template[19], and annotate the entry point in Kate, in the function “`void TimeDatePluginView::slotInsertTimeDate()`” of the plugin. This function is called every time the user selects the menu element corresponding to the current plugin. We chose this point as the implantation point for allowing the user to chose whenever wants to generate the call graphs. The annotation added in the host is: “`void TimeDatePluginView::slotInsertTimeDate(){ /*IP */`” .

For the `cflow` donor, the desired functionality is to transplant the tree form of the call graphs. Thus, we label the function “`tree_output()`” as the organ entry in the donor system. The organ generates the call graph of a C program and displays it in the tree format (option “`-T`” from cflow). The annotation is: “`void tree_output(){ /*OE*/`”. For the `indent` donor, the desired functionality is to enable Kate to completely format a C source file. We want to format the current opened document

[16]http://kate-editor.org
[17]http://www.gnu.org/software/cflow/
[18]http://www.gnu.org/software/indent/
[19]https://techbase.kde.org/Development/Tutorials/Kate/KTextEditor_Plugins

in the Kate's main page. Thus, we label the function "`indent_single_file`" as the organ entry point. This function reformats the source file, according to the settings of GNU `indent`. The annotation is: "`exit_values_ty indent_single_file(BOOLEAN using_stdin){/*OE*/`" . Another possible transplant from GNU indent would be the indentation of the code as it is being edited. Such transplant would require a different implantation point in the part of Kate's code base that gets executed when an user of Kate types text in the editor. The software engineer need only provide μSCALPEL with these simple annotations and suitable test cases and the reminder of the transplantation process is entirely automated.

We used several different test suites to validate our transplant. First, we execute the original regression test suite, available with `Kate`. Since this test suite does not execute the organ, we augmented it with test cases specifically aimed at executing the organ. Second, we generated an acceptance test suite, aimed at checking the transplanted functionality when executed in the host. As with all approaches to GP, the test cases are used to guide the search for suitable code.

Provision of these test cases remains the responsibility of the software engineer. Such tests, or a large subset thereof, would be likely required by a human transplantation process in any case, so this is not a significant additional burden. Furthermore, even were such costs attributed to the autotransplantation process, it would be likely easier, in many cases, to define suitable test cases to *check* a transplant than it would be to *generate* one from scratch while ensuring it is sufficiently tested. In order to estimate the human cost, we recorded the elapsed developer time required to construct the isolation, acceptance, and regression++ test cases that are specifically required to validate the transplantation, thereby providing an upper bound on human effort. Our estimation for annotations and the test suites is one hour for both of the transplants.

For all our test suites we provide coverage information. Table 4.4a shows the results of our test for `cflow` donor, while Table 4.4b shows the results of our tests for `indent` donor. We also manually generated the ice-box test suite, used by the GP in the process of transplanting the feature.

4.11.2 Experiments and Results

Table 4.4a and Table 4.4b presents the number of runs in which for every test suite, all test cases pass. We report the number of successful runs for the regression, augmented regression, and acceptance test suites individually and also report the coverage achieved by test cases (of the entire `Kate` system, and of just the transplanted organ). This coverage data is that reported by the publicly available coverage metric tool `gcov`. Table 4.4a shows the results of the test suites for `cflow` donor transplant, while Table 4.4b shows the results of the test suites for `indent` donor transplant. For `cflow`, 16 out of 20 runs where unanimously successful, while for `indent` 18 out of 20 runs were unanimously successful.

We deem a transplantation attempt to be successful if (and only if) all the test cases from the corresponding test suite passed. The row labelled "Unanimously" reports the number of transplantation attempts in which *all* test cases passed in all test suites. The line "Isolation" reports the results of the

Table 4.3: Runtime data, averaged over 20 runs.

Donor	Time(min.)	
	Avg	Std Dev
GNU `cflow`	101	31
GNU `Indent`	31	6

isolation test suite, which is used by the GP algorithm for evolving the organ (as opposed to being used for valuation purposes).

Observe that even were we to find that automated transplantation was only successful one a few of the 20 attempts, then this would be sufficient to demonstrate the feasibility of autotransplantation in general. The testing process can be used to validate any transplantation attempt, allowing the software engineer to discard any and all failed attempts. As a result show, autotransplantation achieves a much higher success rate than this, minimal, feasibility requirement. Overall, we have evidence that autotransplantation is feasible for the popular real world system `Kate`. We now turn to the specific research questions, we posed to answer them.

RQ1 Table 4.4a and Table 4.4b revels that for both of the transplants, all the regression test cases passed. However, the organs were not executed by the existing regression test suites, so we manually augmented them to generate the regression++ test suites. Organ coverage for the regression++ test suites is: 59% for `cflow`, and 58% for `indent`. For `cflow` 17 out of 20 transplantation attempts passed all test in these augmented test suites, while for `indent`, 18 out of 20 pass all. Clearly one can never do enough regression testing, but these results provide release some confidence. In future work we plan to use automated search based software testing [125] to further improve autotransplantation regression testing.

RQ2 For `cflow` 18 out of 20 transplants passed all acceptance tests, while for `indent` 19 out of 20 pass, giving confidence that μSCALPEL has successfully transplanted code, such that the desired functionality is available to host program.

RQ3 Table 4.3 reports the timing information for the transplants. On average, transplanting the call graph computation from `cflow` took 101 minutes, while the layout feature from `indent` took 31 minutes. In less than 44 hours total time, we were able to complete all 40 repetitions of the two experiments. The human effort required to incorporate these two new features would surely have been considerably greater.

A Flavour for the Transplants Produced by μSCALPEL Figure 4.19 provides a flavour of the `indent` transplant. Figure 1a shows portion of the vein, identified in the static slicing processing. The vein starts at the function `main`, and ends at organ entry; the function `indent_single_file`. The vein contains the function `process_args`, which initialises globals, based on the command line parameters originally used in the donor `indent`. Figure 1b shows the resulting code after the inlining process.

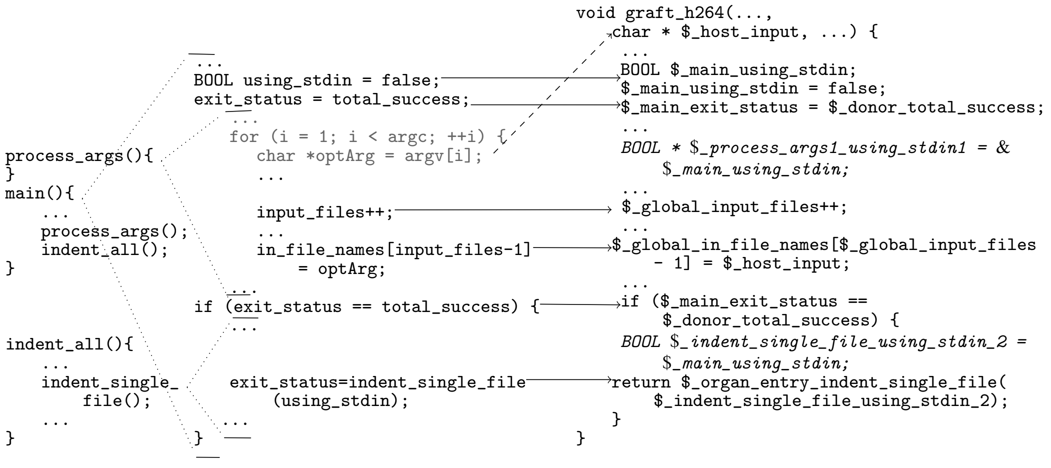
Table 4.4: Transplantation results. Figures marked with * exclude regression test cases that failed before the transplantation (only one for `Kate`).

(a) GNU `cflow` Donor

Category	Pass Rate	Coverage (%)	
		All	Organ
Unanimously	16	-	-
Isolation	18	-	-
Regression	20*	62	0
Regression++	17	74	59
Acceptance	18	52	59

(b) GNU `Indent` Donor

Category	Pass Rate	Coverage (%)	
		All	Organ
Unanimously	18	-	-
Isolation	19	-	-
Regression	20*	66	0
Regression++	18	78	68
Acceptance	19	48	68



```
process_args(){
}
main(){
   ...
   process_args();
   indent_all();
}

indent_all(){
   ...
   indent_single_
       file();
   ...
}
```

```
...
BOOL using_stdin = false;
exit_status = total_success;
   ...
   for (i = 1; i < argc; ++i) {
     char *optArg = argv[i];
     ...
     input_files++;
     ...
     in_file_names[input_files-1]
         = optArg;
   ...
if (exit_status == total_success) {
   ...
   exit_status=indent_single_file
       (using_stdin);
   ...
}
```

```
void graft_h264(...,
   char * $_host_input, ...) {
   ...
   BOOL $_main_using_stdin;
   $_main_using_stdin = false;
   $_main_exit_status = $_donor_total_success;
   ...
   BOOL * $_process_args1_using_stdin1 = &
        $_main_using_stdin;
   ...
   $_global_input_files++;
   ...
   $_global_in_file_names[$_global_input_files
        - 1] = $_host_input;
   ...
   if ($_main_exit_status ==
        $_donor_total_success) {
   BOOL $_indent_single_file_using_stdin_2 =
        $_main_using_stdin;
   return $_organ_entry_indent_single_file(
        $_indent_single_file_using_stdin_2);
   }
}
```

(a) Indent Code (b) Indent Inlined Source Code (c) Transplanted Source Code

Figure 4.19: Transplant operation in Cflow donor transplant. Code snippet from the beginning of the graft. ⋯| means function inlining; `optArg` is mapped to `$_host_input`; ⟶ means original statement replacement under α — renaming; grayed statements are deleted.

The brackets capture the code corresponding to each original function. Figure 1c shows the code transplanted into `Kate` by one of the successful transplants. An α–renaming scheme is used to avoid namespace conflicts within the host, and between the inlined functions.

Some organ statements must be removed, due to failed test cases or incorrect binding to host variables. Some variables may even be unbindable, leading to an uncompilable (or crashable) transplant. For example, the variables `argc` and `argv` simply cannot be bound to host variables, because `Kate` has no concept of 'command line argument'. Fortunately, GP discovers such issues. It removes the first `for` statement in the figure Figure 1b . The variable `optArg` is used for parsing the command line parameters of `indent`. This variable was mapped at the variable `$_host_input$`, thereby correctly using input from `Kate` call graph computation.

Figure 4.20 shows the differences between the original code and the code transplanted in `Kate` in the case of `Indent` donor. The first two additions are done because of the inlining. We generate fresh variables for the actuals of a function call. The first if is removed by the GP, because the value of `lev` is equal with the one of `level_mark_size`. Because of this, the predicate is always true, and so the

```
I  <<output()>>...
I  <<set_level_mark(0,0)>>
+  int $_set_level_mark1_lev1_1 = 0;
+  int $_set_level_mark1_mark1_1 = 0;
-  if (lev >= level_mark_size) {
α   $_global_level_mark_size += $_global_level_mark_incr;
α   $_global_level_mark = xrealloc ($_global_level_mark,
                 $_global_level_mark_size);
-  }
α $_global_level_mark[$_set_level_mark1_lev1_1] =
          $_set_level_mark1_mark1_1;...
/  int $_include_symbol_type;
/  $_include_symbol_type = 0;
       ...
-  if (print_option & PRINT_TREE) {
OE  return $_organ_entry_tree_output ();
```

Figure 4.20: Diff of code transplanted from GNU `cflow`. 'I' means inlining, '+' means line addition, "α" means $\alpha - renaming$, '/' means initialisation separation, while '`OE`' means the return of the organ entry. All the lines with '-' are in the namespace of the original Donor.

`if` statement is not required. In the case of the second `if`, GP removed that statement because the variable `print_option` was not bound. Having this `if` would have caused the individual to fail the test cases. The desired feature is the `tree_output` function, and so, removing the `if` makes the individual successful.

4.12 Conclusion

This chapter introduced μTrans, our approach to autotransplantation that combines static and dynamic analysis to extract, modify, and transplant code from a donor system into a host. We have sought to automatically transplant genuine, non-trivial functionality from one system, called the donor, into an entirely different system, called the host, and validate that the result passes regression testing, augmented regression testing and acceptance testing of the. All these tests are aimed at testing the integration of the transplanted organ in the host system, by the means of not introducing regression faults, and of implementing the desired functionality, as specified by the user of μSCALPEL.

Autotransplantation is a new (and challenging) problem for software engineering research, so we did not expect that all transplantation attempts would succeed. In our empirical study, we systematically autotransplanted five donors into three hosts. 12 out of these 15 experiments contain at least one successful run. μSCALPEL successfully autotransplanted 4 out of 5 features, and all 5 passed acceptance tests. In all, 65% (188/300) runs pass all tests in all test suites, giving 63% unanimous pass rate; considering only acceptance tests, the success rate jumps to 85% (256/300).

We showed that autotransplantation is already very useful. Our first case study compared the autotransplantation of adding support for a new media format to the VLC media player, a task humans had previously accomplished. This study indicates that automated transplantation can be useful, since we showed μSCALPEL could insert support for H.264 encoding into the VLC media player in 26 hours, passing all regression and acceptance tests. Our second case study automatically transplanted 2 functionalities that were missing from Kate, and requested by the users of Kate, on Kate's development forum. μSCALPEL required on average 101 minutes for transplanting the call graph generation feature, while transplanting the C layout feature took on average 31 minutes. The success rate for `cflow` donor is 80% (16/20), while the success rate for `Indent` donor is 90% (18/20). Our evaluation provides evidence to support the claim that automated transplantation is a feasible and, indeed, promising new research direction. μSCALPEL is open source and public available at `http://crest.cs.ucl.ac.uk/autotransplantation`. The new version of μSCALPEL, based one ROSE compiler infrastructure is to be released soon, as an open source project.

Chapter 5

Multilingual Software Transplantation

Despite recent advances in automated software transplantation (Chapter 2), all previous work has been confined to a single language: the transplanted code comes from a donor system that is written in the same language as the host system into which the code is transplanted. Chapter 4 describes our approach for monolingual transplantation, but what if the languages of the host and of the donor differ? This chapter propose solutions for the problem of multilingual software transplantation (*MultiST*). Multilingual transplantation is the more general problem of monolingual software transplantation, tackled in the first part of the thesis (Chapter 4), in which the languages of the donor and host differ. We implemented our approach in τSCALPEL and report an empirical study and three case studies where τSCALPEL transplants 10 organs between non-trivial, real-world programs written in different programming languages.

This chapter is organized at follows: Section 5.1 introduces the problem of multilingual software transplantation; Section 5.2 shows a motivating example for MultiST; Section 5.3 formally define the problem of multilingual software transplantation using the formalism introduced in Section 4.3; Section 5.4 reports our implementation for MultiST; Section 5.5 evaluates our MultiST approach; Section 5.6 presents three case studies; Section 5.7 discusses the maintainability of multilingual software transplants; and Section 5.8 concludes.

5.1 Introduction

Our approach for multilingual software transplantation is a novel hybrid combination of rule-based translation [282], augmented by genetic programming [167], for organ translation and synthesis. It applies rule-based translation to the core constructs of a language, leaving holes for the non-core constructs, like API calls. These holes are, by construction, well-suited for synthesis, because they are small and equipped with a specification: those parts of the organ the holes replaced.

We introduce τTrans, our algorithm, and approach, for multilingual software transplantation (Section 5.3), and τSCALPEL (Section 5.4), our implementation of τTrans. We evaluate τSCALPEL in ten separate transplantation experiments over two different language pairs. The first five of these are transplants from `Java` donor programs, into Swift host programs while the second five are from `Fortran` donor programs, into `Python` hosts.

We chose the first scenario, from `Java` donors to Swift hosts, since this demonstrates a practical use-case scenario for this technique; permitting code developed for the Android platform to be automatically transplanted into code developed on the iOS platform. In this way we illustrate, through our choice of empirical study subjects, the actionability of this research agenda; it tackles platform fragmentation in the rapidly-expanding mobile development space.

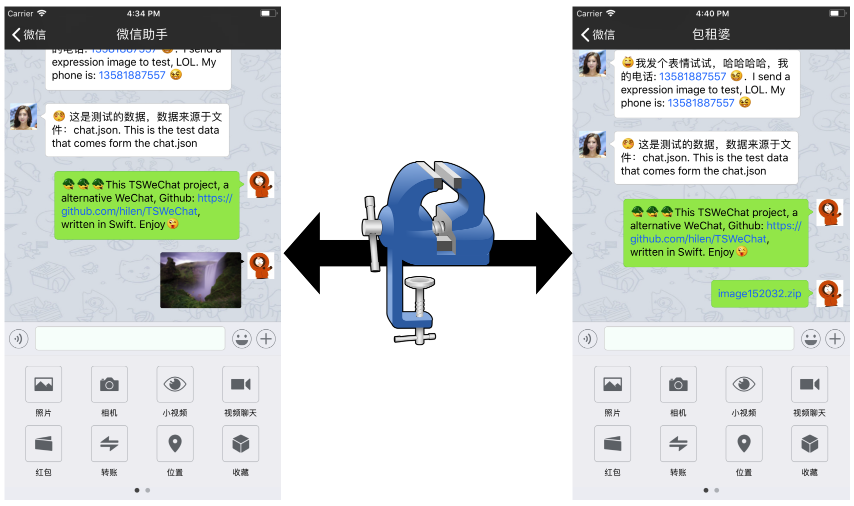
To motivate the need for automated support in this space, we manually examined[1] the functionality offered by the 50 most popular apps in the iOS App Store; 42 out of these also exist in Google Play app store. In each case, app developers are reimplementing near-identical functionality for each platform. The cost of developing each such app has been estimated to reside in the range \$0.5–\$1m [348], suggesting that duplication is a highly non–trivial cost; one that would be better avoided where possible. τSCALPEL ameliorates the cost of reimplementation by automating the transplantation of functionality across programming languages and platforms.

We chose the second scenario, from `Fortran` donors to `Python` hosts, to illustrate the way in which multilingual transplantation can extract useful functionality from the oldest high-level language available into a more recent language, allowing our transplantation approach to revive code, and the functionality it carries, across the decades and over several generations of programmers. There is much untapped functionality in legacy code, such as carefully-implemented and repeatedly-tested mathematical routines written in Fortran [50]. Such code may be written in an out-of-date languages, but it has passed the test of time. Sadly, such tried and tested code cannot be exploited in more recent programs, written in Python, for example, without considerable developer effort. It might even be difficult to *find* programmers who are fluent in both languages[2].

Our results are highly encouraging: we run each of the ten transplants in the empirical study (Section 5.5) 20 times to collect sufficient data for robust analysis. Out of these 200 different runs, 145 are completely successful transplants, passing all tests cases, regression and acceptance test, where tests have been specifically augmented to cover the transplanted functionality. τSCALPEL is also computationally efficient: it required only 33 hours for 10 different transplantations, averaged over 20 runs.

We also report 10 case studies in more detail, comparing all of our organs with human-written implementations in the language of the host (Section 5.6). Comparing against human-written alternatives allows us to investigate the correct behaviour of the transplanted code against a much tougher test criterion. That is, we generated 10,000 test cases from two of the human alternatives and 100 test cases from the third one, and use these to thoroughly test the transplanted code, using the human-written alternative as a reference implementation that provides a reliable oracle [31].

[1]Data extraction performed on 5th February 2018.
[2]https://www.geek.com/news/nasa-seeks-programmer-fluent-in-60-year-old-languages-to-work-on-voyager-1638276/



expression image to test, LOL. My phone is: 13581887557

这是测试的数据，数据来源于文件：chat.json. This is the test data that comes form the chat.json

This TSWeChat project, a alternative WeChat, Github: https://github.com/hilen/TSWeChat, written in Swift. Enjoy

我发个表情试试，哈哈哈哈，我的电话: 13581887557 . I send a expression image to test, LOL. My phone is: 13581887557

这是测试的数据，数据来源于文件：chat.json. This is the test data that comes form the chat.json

This TSWeChat project, a alternative WeChat, Github: https://github.com/hilen/TSWeChat, written in Swift. Enjoy

image152032.zip

Figure 5.1: After successful multilingual software transplantation, `TsWeChat` can send and receive compressed files[3].

These case studies further confirm that our transplanted organs produce equivalent results in their hosts, while the observation that human developers cared sufficiently about implementing their functionality provides additional evidence for research actionability.

The main contribution of this chapter are:

1. The formalization of the multilingual software transplantation problem: software transplantation between programs written in different languages;
2. τSCALPEL, the implementation of multilingual software transplantation;
3. The evaluation of τSCALPEL: we transplanted 10 real world organs between 10 hosts and 10 donors written in different languages, yielding a success rate of 82% for Java to Swift and 63% for Fortran to Python transplants.
4. Detailed examination of 3 examples, comparing with human-written alternative (in the language of the host) as a further demonstration of both correct behaviour of the transplanted functionality and its applicability.

5.2 Motivating Example

This section presents the running examples for τSCALPEL, our tool for multilingual software transplantation. This example is drawn from one of the experiments in the empirical study: the compress organ transplantation from the `Compress` donor, into the `TSWeChat` host program.

`TSWeChat` is an iOS chat application that supports sending files and is written in Swift. `TSWeChat` does not support file compression, so it wastes bandwidth when sending files. Perhaps we can find a suitable compressor and transplant it into `TSWeChat`? `Compress` is a Java library that supports file compression. Transplanting the compression functionality `Compress` into `TSWeChat` is

[3]The vise is licensed for "noncomerical reuse" and published at `https://cdn.pixabay.com/photo/2015/05/29/05/01/clamp-788906_960_720.png`, visited on 29.10.2018

challenging, because we need to identify all the code that implements compression in the donor that has 24k LOCs. Further, we need to fix all the issues that arise when moving the code into the host (*e.g.* namespace conflicts), and connect the program status of the transplanted code with `TSWeChat` at the implantation point. On top of these challenges is the fact that the two programs are written in different languages, vastly complicating all phases of the transplanting `Compress`'s functionality into `TSWeChat`.

τSCALPEL solves all of these problems: it automatically transplants *and translates* functionality from a donor written in one language into a host written in another; in this case from `Compress` in Java to `TSWeChat` in Swift. To use τSCALPEL, a user must identify where to implant the desired functionality into the host and a donor program that implements the desired functionality, then annotate the implantation point in the host and the entry point of the desired functionality, or organ, in the donor.

The organ entry point is the method `process` in the file `LZF.java` at line `30`. The implantation point in `TSWeChat` is in the file `TSChatViewController+Interaction.swift` at line 95, in the method `resizeAndSend`. The method `resizeAndSend` is triggered when a user sends an image. Finally, the user must provide an organ test suite that is written in the language of the host program and that exercises the functionality to be translated and transplanted.

Given these inputs, τSCALPEL extracts the organ from the donor, translates what it can using a small set of rules, then synthesises code to fill in the holes left by the code it could not translate. τSCALPEL successfully transplants and translates the compression functionality into `TSWeChat`. The second row in Table 5.3 of Section 5.5 reports the results of this multilingual transplantation.

Figure 5.1 shows the results of τSCALPEL, when transplanting compression functionality, from `Compress` into `TSWeChat`[4]. Before τSCALPEL was applied, `TSWeChat` is unable to compress the image. Afterwards, `TSWeChat` compresses it and sends the compressed image (`image152032.zip` in the right image of Figure 5.1). To decompress the now-compressed image, we rely on iOS's built-in support for file decompression: after the compression transplantation, when `TSWeChat` receives a compressed file, iOS' file browser presents it to the user, who can click to decompress it. It is the fact that TSWeChat uses iOS to receive the files that permitted us to only transplant compression and not decompression when updating TSWeChat with the ability to send and receive compressed files.

5.3 Problem Definition

Chapter 4 tackles the problem of monolingual software transplantation: transplantation from a donor to a host program, both written in the same language. *Multilingual Software Transplantation* (MST) moves software from a donor to a host program, when the host and the donor programs are written in different programming languages.

[4]We used TSWeChat, revision 57c0d48, accessed on Jan. 2016, which you can find at `https://github.com/hilen/TSWeChat`.

Section 4.3 defines the transplantation formalism that we are going to use in this chapter as well. In terms of specification, MST is a straightforward realization of Figure 4.9. The difference between monolingual and multilingual software transplantation is the transplantation process itself, $\mathscr{T}$. In monolingual software transplantation, $\mathscr{T}$ moves lines of source code, applies alpha-renaming to avoid namespace conflicts, constructs α_i by moving a vein into the host at the implantation point $\square$. In the case of MST, $\mathscr{T}$ must also translate the code from the language of the donor into the language of the host.

Our approach to MST rests on the insight that language families share *core* constructs. Core constructs have (nearly) equivalent semantics that appear in many, even most, languages in a language family, with possibly different syntax; in aggregate, they define a language family. In the imperative family, two examples are assignment or if statements. The rest of the code is *non-core*, consisting of unusual language constructs, or domain- or project- specific APIs. Core constructs are less likely to undergo fundamental change, either syntactic or semantic (as each language evolves), especially compared to non-core code.

Rule-based translators must chase their source and target languages as they evolve, so keeping them relevant is a recurrent work. Synthesis struggles with lack of specifications and filling large holes. Our solution to MST combines them so that each addresses the weakness of the other: τTrans restricts rules to core constructs to mitigate effort of maintaining the rules. Its partial translation builds a scaffolding that leaves behind small holes equipped with IO oracles, thus addressing synthesis' specification problem. Synthesis, in turn, automatically keeps τTrans up-to-date with a pair of languages as they evolve.

τTrans realizes MST in two stages. In the first stage, we use rules to translate all core constructs and replace all noncore constructs with holes. In the second, we use genetic programming to synthesise code to fill the holes. Each hole, created during rule-based translation, shares inputs and outputs with the noncore construct $\bar{c}$ it replaced in O_D. After de-sugaring to handle noncore language constructs, holes correspond to single function or API calls. Each hole's inputs and outputs constrain the search for candidates to fill it, especially when type annotations are available. Crucially, $\bar{c}$ captures its hole's specification in the form of an IO oracle (the code to $\bar{c}$ is essentially a kind of 'test harness'). Thus, MST gives rise to a naturally occurring code synthesis problem: that of filling a hole in O'_H using its $\bar{c}$. In one stroke, it ameliorates two problems that bedevil code synthesis: the lack of specifications and the problem of handling large holes.

For us, language constructs are statements, type annotations, operators, or function calls. Operationally, we define core language constructs to be those constructs for which we have a translation rule; all others are noncore. We map noncore statements to function calls. A *hole* uniquely identifies a noncore identifier in the organ. Because holes replace identifiers, the same hole might appear in multiple places. To preserve correctness, all of a hole's apperances must be filled by the same thing; filling a hole is a simultaneous substitution across all its appearances.

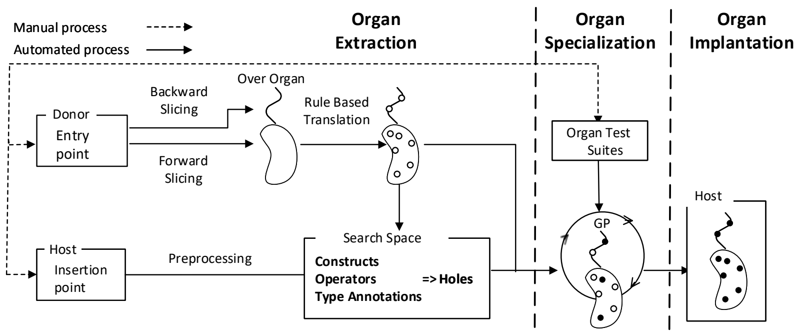


Figure 5.2: The architecture of τTrans: *Organ extraction* extracts a partial-translated organ with holes; *Organ specialization* fills in those holes and inserts any needed input or output adapters the host needs to interact with the organ; and *Organ implantation* copies the translated and adapted organ into H at I_P.

The type context $\Gamma = \{e_i : \tau_j\}$ maps all the operators, functions, and variables (e_i) to their types (τ_j). To handle *holes*, we overload $\Box$ (initially meaning the missing functionality from H) to denote holes as defined above and extend the range of Γ: $\Gamma_\Box = \Gamma \cup \{\Box_i : \Box_j \mid i \neq j\}$, where $\Box_i$ is an unknown operator, function, or variable name and $\Box_j$ is an unknown type annotation. For us, an *abstract syntax tree* of an organ (AST) then becomes AST $\triangleq (V, E, l)$, where $l : V \rightarrow \Gamma_\Box$ labels the nodes in the AST: The key adaptation of the AST for MST we make here is to augment the range of l with holes $\Box_i$ to represent unknown labels. An *iceboxed organ* (IxO) is an AST with holes. $\Gamma_\Box$ is implicit to a top-down traversal of the IxO, so every hole identifying a function, operator, variable name, or type annotation in the AST maps to a hole in $\Gamma_\Box$ and vice versa. We need a top-down traversal to determine identifer scope.

The next step in MST is to fill the holes to create a hole-free AST from which a compiler can generate code. The search space of hole-filling candidates comprises variables and functions (including operators and APIs) in scope at the host, in the transplanted donor vein, in a library of components, or their composition. The inputs and outputs of each hole constain this space. As noted above, a hole's corresponding noncore construct in the donor provides an IO oracle that further constrains the search space.

Once we have filled all the holes, we implant O'_D into H to create the postoperative H'. We do not construct α_o, an output adapter, and instead require the user to insure that $Q_D \Rightarrow Q_H$. Finally, we use test cases to assesses Q_H: we consider H' to be *sufficiently* correct if it passes its integration tests.

5.4 Genetic Multilingual Transplantation

In the previous chapter (Chapter 4) we tackled the problem of monolingual software transplantation: transplantation from a donor to a host program, both written in the same language [30]. *Multilingual Software Transplantation* (MultiST) moves software from a donor to a host program, when the host and the donor programs are written in different programming languages.

```
for (int i = 0; i < threadCount; i++) {
    int wordsForThread = wordsPerThread;
    if (additionalWordsPerThread > 0) {
        wordsForThread++;
        additionalWordsPerThread--;
    }
    WordSorter sorter = new
         WordSorter(words,
         currentOffset, currentOffset +
         wordsForThread);
    wordHandlers.add(sorter);
    currentOffset += wordsForThread;
}
```

Figure 5.3: Sort.java

```
for var i : ⟨□0⟩ = ⟨□1⟩; i <
     threadCount; i++ {
    var wordsForThread : ⟨□2⟩ =
         wordsPerThread;
    if (additionalWordsPerThread > ⟨□3⟩)
        {
        wordsForThread++;
        additionalWordsPerThread--;
    }
    var sorter ⟨□4⟩ = ⟨□5⟩(⟨□6⟩ : words,
         ⟨□7⟩ : currentOffset, ⟨□8⟩ :
         currentOffset +
         wordsForThread);
    wordHandlers.⟨□9⟩(sorter);
    currentOffset += wordsForThread;
}
```

Figure 5.4: Sort.java after our rule-based translation system.

5.4.1 The Architecture of τSCALPEL

Figure 5.2 shows the overall architecture of τTrans. Organ extraction creates an over-approximation of the actual organ that we call the over organ. Organ specialization translates an over organ into an host's language and adapts it to the implantation point in the host. Finally, organ implantation copies the translated and adapted organ into the host, forming the postoperative host.

Organ Extraction: The first stage of MultiST is organ extraction. Similar with MonoST [30], we use forward and backward slicing to extract the over organ, the over-approximation of the actual organ. In practice, this will first extract the organ entry point method. We use backward slicing to extract a path from the entry point of the donor that reaches the organ entry (called vein in previous work [30]) that is the entry point of the code transplant. The forward slicing will extract the surrounding class of the organ entry, as well as the transitive closure of all the classes and methods from the donor's source code that can be reached either from the vein or from the organ entry function. In the extraction process, we keep the folder structure of the donor program. Thus, the over organ that the organ extraction produces, contains many files, each containing potentially more classes and methods. Before going to the next stage, we also translate the over organ process that Section 5.4.2 details.

Organ Specialization: The organ specialization step has two purposes: 1) it fills the holes from the over organ; and 2) it reduces the over organ overapproximation, by selecting which lines from the organ are actually required for passing the organ's test suite.

Organ Implantation: is the final stage of MultiST. This is the simplest stage of MultiST and involves just adding a call at the implantation point in the host to the organ's entry point.

τTrans requires a rule-based translator for each donor–host language pair that it supports. We used ANTLR to implement these translators. They traverse the organ, translate core constructs, and replace noncore with uniquely tagged holes for each identifier. We implemented translators for Java to Swift; and Fortran to Python.

For us, language constructs are statements, type annotations, operators, or function calls. Operationally, we define core language constructs to be those constructs for which we have a translation rule; all others are noncore. We map noncore statements to function calls. A *hole* uniquely identifies a noncore identifier in the organ. Because holes replace identifiers, the same hole might appear in multiple places. To preserve correctness, all of a hole's apperances must be filled by the same thing; filling a hole is a simultaneous substitution across all its appearances.

τTrans is a two stagial approach. In the first stage, we use rules to translate all core constructs and replace all noncore constructs with holes. In the second (Section 5.4.3), we use genetic programming to synthesise code to fill the holes. Each hole, created during rule-based translation, shares inputs and outputs with the noncore construct $\bar{c}$ it replaced in the organ. After de-sugaring to handle noncore language constructs, holes correspond to single function or API calls. Each hole's inputs and outputs constrain the search for candidates to fill it, especially when type annotations are available. Crucially, $\bar{c}$ captures its hole's specification in the form of an IO oracle (the code to $\bar{c}$ is essentially a kind of 'test harness'). Thus, MultiST gives rise to a naturally occurring code synthesis problem: that of filling a hole in the organ using its $\bar{c}$ as defined in the donor. In one stroke, it ameliorates two problems that bedevil code synthesis: the lack of specifications and the problem of handling large holes.

5.4.2 MultiST Rule Based Translation System

To see how the first stage of our MultiST approach works, consider the code from `Sort.java` in Figure 5.3. Figure 5.4 shows the over organ formed from the code in Figure 5.3 after applying our rule-based translator from `Java` to `Swift`. Dark gray represents rule-based translations, light gray represents code simply copied over, *i.e.* identity translations, and $\square$ represents holes to be filed in the second stage.

Our approach to MultiST rests on the insight that language families share *core* constructs. Core constructs have (nearly) equivalent semantics that appear in many, even most, languages in a language family, with possibly different syntax; in aggregate, they define a language family. In the imperative family, two examples are assignment or if statements. The rest of the code is *non-core*, consisting of unusual language constructs, or domain- or project- specific APIs. Core constructs are less likely to undergo fundamental change, either syntactic or semantic (as each language evolves), especially compared to non-core code. Rule-based translators must chase their source and target languages as they evolve, so keeping them relevant is a recurrent work. Synthesis struggles with lack of specifications and filling large holes. Our solution to MultiST combines them so that each addresses the weakness of the other: τTrans restricts rules to core constructs to mitigate effort of maintaining the rules. Its partial translation builds a scaffolding that leaves behind small holes equipped with IO oracles, thus addressing synthesis' specification problem. Synthesis, in turn, automatically keeps τTrans up-to-date with a pair of languages as they evolve.

```
1   public class Java2SwiftTranslatorVisitor extends TranslatorVisitorBaseClass {
2
3       Override
4       public String visitBasicForStatement(BasicForStatementContext ctx) {
5           String result = "";
6           for (int i = 0; i < ctx.getChildCount(); i++) {
7               if (ctx.getChild(i).getText().equals("for") ||
                    ctx.getChild(i).getText().equals(";")) {
8                   result += ctx.getChild(i).getText() + "␣";
9               } else if (ctx.getChild(i).getText().equals("(") ||
                    ctx.getChild(i).getText().equals(")")) {
10
11              } else if (ctx.getChild(i)) instanceof StatementContext) {
12                  currentBlockStm = ((StatementContext)
                        ctx.getChild(i)).statementWithoutTrailingSubstatement().block();
13                  result += "␣" + visitBlock(currentBlockStm, Constants.TemplateBegin +
                        Constants.forStatementAnnotation
14                          + indexForStatement + Constants.TemplateEnd) + "␣";
15              } else {
16                  result += "␣" + visit(ctx.getChild(i)) + "␣";
17              }
18          }
19          return result;
20      }
```

Figure 5.5: τSCALPEL's translation rule for a basic for statement.

In practice, we implement our rule-based translation system as ANTLR parsers. For this, we extend the class `TranslatorVisitorBaseClass` to generate `swift` code from `java` code for example. For example, one of our translation rule for the basic `for` statement between `Java` and `Swift` is:

The translation rule in Figure 5.5 translates a basic for statement from `java` into `swift`. Here we pass the list of the children of the `BasicForStatementContext` that represents a `for` statement in `java`. Its children can be the texts of `for`, `;`, `(`, or `)`. In this cases, we append the texts for the keyword `for`, but we do not output nothing for the parenthesis, as they do not appear in the `swift` language.

The interesting case in this translation rule is that of a `StatementContext`. The `StatementContext` is either the for initialize statement, or its block. In Figure 5.3, both the init of for (line 169) and its block(lines 170 to 178) are `StatementContext`s. When we reach such case, we add a annotation for the current statement context, as we see at line 13 in Figure 5.5. The type signature of `visitBlock` is: `public String visitBlock(BlockContext ctx, String annotation)`. It's implementation will append the annotation in the output string, such that we can uniquely identify and refer to each hole that will appear in that block. For example, the constants that we use in Figure 5.5 to mark a block hole are:

```
public final class Constants {
    public static final char TemplateBegin = 'TODO{cannot␣put␣here␣the␣special␣character}';
    public static final char TemplateEnd = 'TODO{cannot␣put␣here␣the␣special␣character}';
    public static final String forStatementAnnotation = "STM_FOR_STATEMENT_";
}
```

We use these constants, together with an unique index of the block (`indexForStatement` at line 13) in the entire input source code to uniquely identify a block, and further to look for it in the rest of our translator. Further, we continue to visit each and every node in the AST of the `java` `for` statement in

Figure 5.3. Each different type of AST node is processed according to the rules of the `Swift` language, such that they become valid code, while having holes for places where we do not know the exact mapping. Such places are language constructs, operators, and type annotations.

When we visit the for initialization statement: `int i = 0; i < threadCount; i++` we first reach the variable declaration `int i = 0`. The rule for visiting variable declaration is:

```
 1  Override
 2  public String visitLocalVariableDeclaration(LocalVariableDeclarationContext ctx) {
 3      String result = "";
 4      String type = visit(ctx.unannType());
 5      /*
 6       * We need to separate the declarations!
 7       */
 8      ArrayList<String> declarations = appendDeclaratorList(ctx.variableDeclaratorList(),
            type);
 9      for (String current : declarations) {
10          result += current;
11      }
12      return result;
13  }
```

This rule is generic and will separate variable declarations on separate line, for an easy translation in case of initializations. In the case of our example declaration in the for initialization, we only have one declaration. At line 4, we visit the type declaration at line 4. The top-bottom visitor, will process its type and add the corresponding hole for the type. For the same type in the source language, we will always have the same hole in the target language. We use a symbol table to guarantee this.

When processing a type, we look in the symbol table of the donor code, to identify types declared in the donor code, for which we use as they are since we have our declarations, and for types that do not appear there, for which we need to add holes (eg. system level APIs). This is the case of `int` for which we add a hole in the visitor of a type declaration ($\langle \Box_0 \rangle$ in Figure 5.3). In case of type holes, we look in all the APIs available in the target language (`swift` in this case).

For this, we implemented an util tool that analyses the Swift binary for such declaration. Our tool is based on the unix tool `nm`(https://linux.die.net/man/1/nm). This tool looks into the function definitions in the target swift binary library and extracts all the functions defined together with the type definitions, by using the debugging symbols. This gives us the full list of APIs defined in the target language.

An interesting case arises when method signatures do not match. For example, in Java, we could have a method `find(String needle, String haystack )`, while in Swift the method could be `find(String haystack, String needle)`. More than this, the number of parameters might also not match. For this, we use the following visitor of a method invocation:

```
1       Override
2       public String visitMethodInvocation(MethodInvocationContext ctx) {
3           String result = "";
4
5           String currentClass = "";
6
7           if (ctx.typeName() != null) {
8
9               VariableDeclarationWithBlockID varDeclWithId = this.symbolTable
10                      .findVariableInSymbolTable(this.reacheableBlocks,
                            ctx.typeName().getText());
11              if (varDeclWithId != null && varDeclWithId.getVarDecl() != null) {
12                  currentClass = TypeWithAPIsAndDataMemberChromosome
13                          .removeAnnotationsKeepArrays(varDeclWithId.getVarDecl().getType());
14              } else {
15                  currentClass = ctx.typeName().getText();
16              }
17          } else {
18              // current class
19              currentClass = this.currentClassName;
20          }
21
22          for (int i = 0; i < ctx.getChildCount(); i++) {
23              if (ctx.getChild(i).getText().equals("(") || ctx.getChild(i).getText().equals(")")
24                      || ctx.getChild(i).getText().equals(".") ||
                            ctx.getChild(i).getText().equals("super")) {
25                  result += ctx.getChild(i).getText();
26              } else {
27                  String ret = visit(ctx.getChild(i));
28                  if (ret == null) {
29                      /*
30                       * This is the identifier for the method name, since we
31                       * return null for terminals!
32                       */
33                      ret = appendAPIsArray(ctx.getChild(i).getText(), currentClass);
34                  }
35                  result += ret;
36              }
37          }
38
39          // append holes for the rest of the symbols defined in the local symbol table
40          result += this.symbolTable.appendLocalSymbols(this.reacheableBlocks)
41
42          return result;
43      }
```

Figure 5.6: τSCALPEL's method invocation visitor that translates a method invocation from Java into Swift

In `visitMethodInvocation`, we first add the type (if defined in our symbol table), or hole at lines 7–20. In the for loop at lines 22–37, we first process the constant literals (lines 23–25) and then we process the variable references (lines 26 – 35). Finally, we add holes at line 40 for all the variables that are available in the local symbol table. This approach allows us to handle cases where the APIs signatures do not match: since everything is a hole in the function argument list, we can match the parameters in any order. All these arguments are optional, so we do not have to provide a mapping for all of them. The genetic synthesis stage will further try to match the correct variables, by using type restrictions.

The next step in MultiST is the *"genetic synthesis"* phase described in Section 5.4.3 below. It's aim is to fill the holes to create a hole-free AST from which a compiler can generate code. The search space of hole-filling candidates comprises variables and functions (including operators and APIs) in scope at the host, in the transplanted donor vein, in a library of components, or their composition. The inputs and outputs of each hole constain this space. As noted above, a hole's corresponding noncore construct in the donor provides an IO oracle that further constrains the search space.

5.4.3 Genetic Synthesis

The genetic synthesis stage of MultiST takes as input the over organ with holes. The purpose of GP is to fill the holes in the multilingual organ to obtain a compilable organ that passes the organ's test suite. GP is the second stage of our two-stage, rule-then-synthesis, organ translation.

In Section 5.4.1, we defined the search space over hole fillings to be constrained by the number and type of the holes inputs and outputs and the IO behavior of the hole's analog in the donor. As an optimization, we partition holes into kinds to further constrain them and speed up search. We consider three kinds of holes: language construct, function, and type annotation. For example, an entry in the hole matching table for holes in Figure 5.4 is:

```
Hole: □0; initial value: int; candidates: int, double, float, ...
Hole: □1; initial value: threadCount; candidates: threads, a, b, ...
```

Above, $\Box_0$ shows the search space restrictions for a data type hole: we match primitive data types to primitive data types. For the int type, the annotation is Swift, we try to map it to the primitive data types in Java. $\Box_1$ is a operator (variable) hole. Here we restrict the space of the possible mappings to the variables available in H at I_P with the same types as the one that we select to fill $\Box_1$: threads, a, b, c.

We prioritize candidates for filling a hole by using the Levenshtein distance [5] between the initial name and each of the candidates. Our intuition is that similar functions, or data members, have similar names across languages. To prioritize a candidate c we compute its score by using the Levenshtein distance. Further, we pick c according to its score. Let d be the Levenshtein distance, N_0 the string value of the hole in D, N_C the string value of the current candidate to fill the hole. We use $|-|$ to denote the length of its string argument. Then we compute the score as:

$$score(c) = 1 - \frac{d(N_0, N_C)}{|N_0|}$$

Chromosomes: τTrans uses two chromosomes for GP. The *Statements chromosome* works as in previous work [30]: it removes lines from the organ to reduce the over-approximation. The *Holes chromosome* is new for MultiST; it stores the current candidates for filling each hole in the over organ. It generalizes the variable matching chromosome [30] by handling k-nary operators and type

[5]https://en.wikipedia.org/wiki/Levenshtein_distance

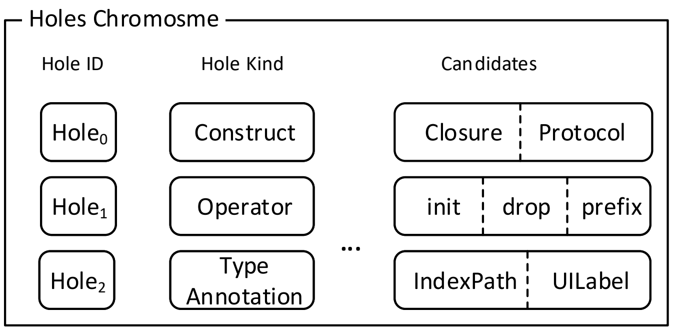


Figure 5.7: τTrans's holes chromosome maps holes to candidates for filling them; each hole has a kind and is restricted by the types of its inputs and outputs.

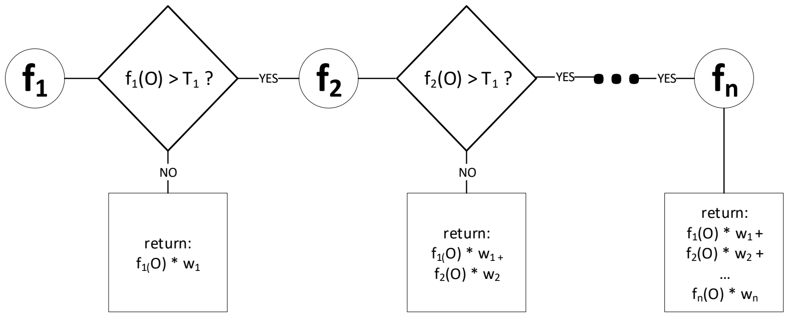

f1
f1(O) > T1 ?
YES
f2
f2(O) > T1 ?
YES
YES
fn
NO
NO
return: f1(O) * w1
return: f1(O) * w1 + f2(O) * w2
return: f1(O) * w1 + f2(O) * w2 + ... fn(O) * wn

Figure 5.8: τTrans's multifunction, phasic fitness function: given the sequence of (sub)fitness functions $f_1, f_2, \cdots, f_n$ where $f_{i+1}(p) \geq T_{i+1} \Rightarrow f_i(p) \geq T_i$ for all programs p, τTrans computes p's global fitness sequentially, in phases: it only continues to f_2 if $f_1(p) \geq T_1$, f_1's fitness threshold, and so on. Otherwise, τTrans returns the current fitness.

annotations, rather than nullary operators as in our previous work [30]. Figure 5.7 shows the different kinds of hole chromosomes.

τTrans uses mutation and crossover operators similar with μSCALPEL [30], extended to the holes chromosome.

5.4.4 Phasic Fitness

GP typically defines its fitness function using test suites. Under this regime, fitness evaluation is computationally intensive. To mitigate this computational cost, GP typically samples tests from the test suite [247]. We use a different approach: along the lines of Langdon *et al.* [178] we use *"phasic" fitness* to reduce the computational cost of fitness evaluations. Figure 5.8 shows our fitness function that is composed of an ordered set of (sub)fitness functions. Naïve evaluation of this phasic fitness function is computationally intensive.

Our solution to improve the performance is phasic fitness evaluation: we order the subfitness functions under two constraints. First, we require $f_j \geq T_j \Rightarrow f_i \geq T_i$, for $i < j$; in other words, if f_j passes its threshold test then f_i does too. Second, we require f_i to cost less than f_j when run from scratch, again for $i < j$. For example, for an individual to pass a test, it must first compile: a subfitness function that tests functionality requires compilation, so passing it implies that a subfitness compilation check would also pass. In addition to compilation, the functionality test does more work: it must run the program and apply a test oracle to it, meeting the second constraint. Here, these subfitness functions can clearly reuse the compilation step. Carefully choosen subfitness functions avoid computationally intensive fitness evaluations for unfit individuals by rejecting them as early as possible, preferably before running the test suite.

In 2015, Qi *et al.* revealed that some GP approaches relied on weak proxies [257]. Weak proxies are test cases in which the fitness evaluation does not check the correctness of candidate programs, but instead checks only its exit code. To avoid it, we manually confirmed that each test case has a test oracle (usually an assertion) that actually checks output. In fact, we exploit *weak proxies* as a subfitness function in our phasic fitness function, to extract signal from test cases that do not pass the test oracle, but otherwise run to completion.

In addition to weak proxies, τTrans considers compiler errors, compiler warnings, unit tests, and strong proxies (where we actually check the output of the test cases). Across all its subfitness functions, τTrans runs the entire test suite at most once. The thresholds t_i are binary for us: we evaluate a (sub)fitness function f_{i+1} only when f_i unanimously passes a test suite.

The pseudocode of our fitness function is:

```
def compute_fitnes(individual: Individual, fitness_components: List[FitnessComponent]) ->
    float:
    fitness_score = 0.0
    for fitness_component in fitness_components:
        (fitness_score_for_component, passes) = fitness_component. compute_fitness(individual)
        if no passes:
            return fitness_score_for_component
        fitness_score += fitness_score_for_component
    return fitness_score
```

The first fitness component that we use is the compilation `fitness_component`. In this component, we try to compile the individual. If the individual does not compile, we penalyse it with -0.1 for every compilation error message that the swift compiler outputs. Once the individual compiles, we give it score 1.0. Next, we look at all the compiler warnings and penalyse the indivudal with 0.01 for each such compiler warning. Note that the score cannot be lower than 0.1: we set a lower bound limit to this for every individual that compiles.

The second fitness component execute the organ's test suite. Here, we devide the component into two parts: correct results to test cases, and test cases that executed without error. Each test cases has a maximum score of $\frac{1}{len(test\_suite)}$. We award 0.2 of it for test case that executes without error, and the rest of 0.8 for the correct test result.

To optimise the execution of the fitness function, as this function involves running the program and can be potentially expensive, we stop the execution as soon as one of the fitness component does not passes. We can do this, as we assume subsumtion order over the list of the fitness component: if the fitness component number i does not pass, there cannot be other fitness component at index j to pass, for all $j > i$.

5.4.5 Implementation

We implemented τTrans in τSCALPEL. τSCALPEL currently supports Java to Swift and Fortran to Python out of the box. Deploying multilingual software transplantation for these languages is straightforward: a developer need only issue

```
java -jar τSCALPEL  --organEntry Account:adjustForExchangeRate --implantationPoint
     SIP/SipCalculatorViewController.swift:206
```

to transplant, for example, `jGnash`'s conversion functionality at `adjustForExchangeRate` in the class `Account` into `SIP-Calculator` at line 206 of `SipCalculatorViewController.swift`.

Supporting another language pair requires implementing a rule-based translator for their shared core, to handle τSCALPEL's first phase. In practice, one can augment the donor's parser or use a source-to-source compiler (*e.g.* TXL [71]). We emphasize that language family's core constructs are, by definition, similar and few in number: for the two pairs we report here, our translators have only 117 translation rules for Java to Swift and 101 translation rules for Fortran to Python. Although the number of rules might seem large, the most of the rules are repetitive. For example, our Java AST visitor handles 11 binary expressions. The translation rules to Swift for these 11 binary expressions are identical, differing only in their operator: *e.g.* addition, multiplication, bit shifts, *etc.*. Repetitions, like this one, drop the number of unique rules to 42 for Java to Swift language pair. Also, one does need to provide rules for all of a pair's core constructs, which, in any case, change over time. Given a translator for the core, τSCALPEL handles the rest, including missing core constructs: its second phase that fills holes using genetic synthesis is language independent.

τSCALPEL computes phasic, multifunction fitness. To define a subfitness, one implements `FitnessComponentInterface`, which declares `getFitness()`. To add a new implementation of this function to τSCALPEL, one appends it to an ordered list[6]. The order of subfitness functions in this list determines the order of evaluation in Figure 5.8. When adding subfitness functions, the developer must respect the two subfitness orderin constraints that we define above.

For a better understanding of τSCALPEL we exemplify how it works in practice on a contrived simple multilingual transplantation experiment. This example contains a complete walk-through of the multilingual software transplantation process to facilitate its understanding and to clarify the manual steps that it involves. All the listings presented here are produced by τSCALPEL. These listings are a simplified version of the outputs that τSCALPEL produced.

Figure 5.9 lists the source code of the host and donor programs. The Swift host program H (Figure 5.9a) simply reads an array of integers from its CLI in the variable `input`. All the arguments to the CLI are appended to the array `input`.

[6]In commit with id ee05a3e of τSCALPEL, this list is realized as the array `listOfComponents` at line 71 in `appendFitnessComponents()` of the class `FitnessFunction`.

The donor program D (Figure 5.9b) consists into a Java class called `donor`. It implements two methods `computeSum` that computes the sum of its input `a`; and `processArray` that makes 0 all the negative elements in its input `a`. The result of it's main function is to compute the sum of all the positive elements in the array. Similarly with H, D reads an array of integers from its arguments, but has an additional functionality that we seek to transplant in H: it computes the sum of the array at line 22 by calling the method `computeSum`. Prior to compute the sum, the donor program calls the method `processArray` at line 21.

To call τSCALPEL on these to program, we issue:

```
java -jar /MST/MuScalpel.jar --transplant /MST/examples/sumTransplant --organEntry
    donor:computeSum --implantationPointFile
    /MST/examples/sumTransplant/Host/src/sources/main.swift --implantationPointLine 2
    --APIExtractor /MST/utils/extractTypeDefs.sh
```

The call to τSCALPEL shows also the way in which we provide the inputs:

- `--transplant` the working folder of the transplant
- `--organEntry` the entry point of the donor, specified as class : method
- `--implantationPointFile` the file that contains the implantation point
- `--implantationPointLine` the line number of the implantation point
- `--APIExtractor` path to a util tool that extracts the symbols from the core language library

The role of the *APIExtractor* is to extract all the symbols available in the core language library of the host program (Swift in our example). τSCALPEL will use this information to try to map symbols, such as APIs, or core objects from the organ, that are in the language of the donor (Java in our current example) to the symbols available in the core library of the target language (Swift in our example) during the language translation process. For example, τSCALPEL will map the array's `length` from Java into Swift array's `count` when translating code from Java to Swift. τSCALPEL knows about the fact that Swift arrays support `count` from the output of the APIExtractor. The APIExtractor that we implemented for Java to Swift transplantations relies on *nm* and *Swift demangler* and we implemented it as the following shell script:

```
#! /bin/bash


DEMANGLER="/usr/local/opt/swift/Swift-3.1.1.xctoolchain/usr/bin/swift-demangle␣-tree-only"


INPUT="/Users/alex/Development/TransplantDemo/MST/utils/libswiftCore.a␣␣
    /Users/alex/Development/TransplantDemo/MST/utils/additionalLib.swift.o"

```

```
OUT=$1
rm -f $OUT


for IN in $INPUT

do
    tempFile="temp.out"
    tempMangledNames="tempMangled.out"
    tempMangledNamesRequired="tempMangledReq"

    rm -f tempMangledNamesRequired
    rm -f tempFile
    rm -f tempMangledNames

    nm $IN > $tempFile



    awk '

␣␣␣␣{

␣␣␣␣␣␣␣␣if($2␣==␣"T")␣{
␣␣␣␣␣␣␣␣␣␣␣␣print␣$3
␣␣␣␣␣␣␣␣}

␣␣␣␣}

␣␣␣␣' $tempFile > $tempMangledNames

    cat $tempMangledNames | grep "_TF\|_TZF" > $tempMangledNamesRequired

    while read -r line;
    do
        $DEMANGLER $line >> $OUT
    done < $tempMangledNamesRequired
done

```

The implantation point specified by the arguments
--implantationPointFile and implantationPointLine is used by τSCALPEL to compute the symbol table available in H. On D's side, the user specifies the function computeSum from the class donoras the organ's entry point with the argument: --organEntry donor:computeSum While deciding where to place them may be difficult, these are the only inputs that τSCALPEL requires.

```swift
func compute(array : [Int]) -> Int {
    return 0
}

var input = [Int]()

for arg in CommandLine.arguments {
    if(Int(arg) != nil){
        input.append(Int(arg)!)
    }
}

print(compute(array : input))

```

(a) The Swift host program.

```java
public class donor{
    public int computeSum (int[] a) {
        int sum;
        sum = 0;
        for (int i = 0; i <a.length; i++) {
            sum = sum + a[i];
        }
        return sum;
    }

    public void processArray (int[] a) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] < 0) {
                a[i] = 0;
            }
        }
    }

    static public int mainFunction(int[] args){
        donor d = new donor();
        d.processArray(args);
        int sum = d.computeSum(args);
        return sum;
    }

}
```

(b) The Java donor program.

Figure 5.9: The host and donor for multilingual software transplantation artificial running example.

Now, suppose that the user wants to transplant the functionality of computing the sum of *all* the elements in the array (*i.e.* not just positive as D does), from D to H.

Organ extraction: the first stage of τSCALPEL consists in the static analysis that constructs the *organ* and the *vein* (V) by looking at the donor's call graph. To construct the vein, we first identify the backward paths on the call graph from the organ entry point (*i.e.* the function `computeSum`) until the donor's entry point that is the function `mainFunction` for the current donor program. As in the case of monolingual software transplantation, our assumption is that each and every of the veins correctly and completely initialize the environment in the donor for the execution of the organ. Thus, we can simply pick one of them uniformly at random.

To identify the organ, we look on the forward call graph, starting from the organ entry function: all the functions that it calls. We extract all the code of the organ and its vein by context-insensitively traverse the donor's call graph and transitively including all the functions called by any function whose definition we reach, and is available in the input code. In multilingual software transplantation, before we start the transplantation process, we also need to translate the over-organ from the language of the donor in the language of the host. The translated over-organ and its vein represent the search space for the *genetic programming* (GP) process that guides the transplantation process.

As in the case of monolingual software transplantation, we inline all the function calls up to the organ entry function, automatically annotate the statements, declarations and compound statements on V, and generate fresh variables. Listing 5.1 shows the obtained organ and vein, as τSCALPEL directly outputs it. In Listing 5.1 we show the results after the application of our rule based translation system: the organ is now in Swift, the target language of this example transplant. Section 5.4.2

details our translation system. In practice, we implement it as ANTLR parsers that output directly the translated organ when parsing the donor code, for the use in the next stage of multilingual software transplantation: organ specialization.

In the *organ specialization* stage, τSCALPEL processes the translated code of the over organ that is now in the language of the host (Swift) and annotated with markers for every hole to uniquely identify it, as the following listing shows:

```
var array_GLOBAL_DECL : [Int] = []


 class CLASS_NAME_donor {

   func FUNCTION_NAME_computeSum ( VAR_DECLARATION_donor_computeSum_0
        Var_Name_a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG : inout
        Array_OF_Java_PRIMITIVE_int ) -> Java_PRIMITIVE_int



   {
      VAR_DECLARATION_donor_computeSum_1 var Var_Name_sum_ClassName_donor_FunctionName_computeSum_2 :
           Java_PRIMITIVE_int

      STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_1
           Var_Name_sum_ClassName_donor_FunctionName_computeSum_2 = 0
      VAR_DECLARATION_donor_computeSum_2 var Var_Name_i_ClassName_donor_FunctionName_computeSum_2 :
           Java_PRIMITIVE_int = 0
      STM_WHILE_STATEMENT_1 while Var_Name_i_ClassName_donor_FunctionName_computeSum_2 <
           Var_Name_a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG
           . Java_data_member_lengthIN_CLASSJava_PRIMITIVE_intIN_CLASS_END {
         STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_2
              Var_Name_sum_ClassName_donor_FunctionName_computeSum_2 =
              Var_Name_sum_ClassName_donor_FunctionName_computeSum_2 +
              Var_Name_a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG [
              Var_Name_i_ClassName_donor_FunctionName_computeSum_2 ]
         Var_Name_i_ClassName_donor_FunctionName_computeSum_2 += 1
      }

      STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_3 return
           Var_Name_sum_ClassName_donor_FunctionName_computeSum_2
   }


   func FUNCTION_NAME_processArray ( VAR_DECLARATION_donor_processArray_3
        Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG : inout
        Array_OF_Java_PRIMITIVE_int )

   {
      VAR_DECLARATION_donor_processArray_4 var Var_Name_i_ClassName_donor_FunctionName_processArray_2 :
           Java_PRIMITIVE_int = 0
      STM_WHILE_STATEMENT_2 while Var_Name_i_ClassName_donor_FunctionName_processArray_2 <
           Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG
           . Java_data_member_lengthIN_CLASSJava_PRIMITIVE_intIN_CLASS_END {
         STM_IF_THEN_ELSE_STATEMENT_0 if( Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG
              [ Var_Name_i_ClassName_donor_FunctionName_processArray_2 ] < 0 )
```

```
                STM_IF_THEN_ELSE_STATEMENT_0{
                    STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_4
                          Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG  [
                          Var_Name_i_ClassName_donor_FunctionName_processArray_2  ] = 0
                STM_IF_THEN_ELSE_STATEMENT_0}

                 Var_Name_i_ClassName_donor_FunctionName_processArray_2  += 1
            }

        }

  }


//END FILE



func organInterface(array_GLOBAL_DECL_param: [Int]) -> Java_PRIMITIVE_int{
    array_GLOBAL_DECL = array_GLOBAL_DECL_param

VAR_DECLARATION_donor_mainFunction_5var Var_Name_args_ClassName_donor_FunctionName_mainFunction_FORMAL_ARG
      : Array_OF_Java_PRIMITIVE_int



VAR_DECLARATION_donor_mainFunction_6var Var_Name_d_ClassName_donor_FunctionName_mainFunction_2  :
      Java_OBJECT_donor  =  Java_OBJECT_donor  ()

STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_5  Var_Name_d_ClassName_donor_FunctionName_mainFunction_2
      . JAVA_METHOD_NAME_processArrayIN_CLASSdonorIN_CLASS_END ( Java_OBJECT_donor_processArray
      _FormalForActual_0  & Var_Name_args_ClassName_donor_FunctionName_mainFunction_FORMAL_ARG  )
VAR_DECLARATION_donor_mainFunction_7var  Var_Name_sum_ClassName_donor_FunctionName_mainFunction_2  :
      Java_PRIMITIVE_int  =  Var_Name_d_ClassName_donor_FunctionName_mainFunction_2
      . JAVA_METHOD_NAME_computeSumIN_CLASSdonorIN_CLASS_END ( Java_OBJECT_donor_computeSum_FormalForActual_0
      & Var_Name_args_ClassName_donor_FunctionName_mainFunction_FORMAL_ARG  )

STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_6return
      Var_Name_sum_ClassName_donor_FunctionName_mainFunction_2



}
```

Listing 5.1: Multilingual transplantation: the processed donor program. This listing captures the automatically generated annotations that are used in the GP stage. The greyed boxed items are annotations internally used by τSCALPEL. The purple boxed items are holes that the GP process will fill.

τSCALPEL uses the annotations at beginning of statements (note that a statement does not necessarily corresponds to a source line) in Listing 5.1 to fill the holes. The annotations are internally used by τSCALPEL to uniquely identify the elements of the donor source code, such as statements and lines. We have 2 high level annotations: variable declarations and statement declaration. The statement declaration annotation also specify which types of statements we are operating on. For example, VAR_DECLARATION_donor_computeSum_1 annotate that the current statement is a variable declaration; and STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_1 annotate the current statement type: statement without trailing substatement (this is the type in the ANTLR grammar that we use for parsing).

The holes in Listing 5.1 will be filed in the GP process. The holes name uniquely identify each hole in the over-organ, as well as it's type:

- variable name holes are in the format: Var_Name_sum_ClassName_donor_FunctionName_computeSum_2. `Var_Name` specifies that this hole is a variable name hole. The rest of the string uniquely identifies the hole by using class name, function name, and an unique id.

- type holes are in the format: Java_PRIMITIVE_int. Here, `Java_PRIMITIVE` specifies that is a primitive type, and `int` is the name of the primitive Java type. For an non primitive type, the annotation looks like: : Java_OBJECT_donor, where `donor` is the name of the Java class.

- operator holes are in the format: JAVA_METHOD_NAME_processArrayIN_CLASSdonorIN_CLASS_END. The first part of the string specifies is a Java operator hole: `JAVA_METHOD_NAME`. Next, we have the initial method name in Java: `_processArrayIN`; and finally we have the class name between annotations in: `IN_CLASSdonorIN_CLASS_END`

Similarly with mono lingual software transplantation, τSCALPEL constructs internally the *Organ Dependency List*, to capture the dependencies of a declaration (on its initialization) or statement on the declarations of the variables that it uses and the dependencies of a control flow statement on its compound statement.

τSCALPEL extracts the variables available in the host at the implantation point and uses this information in the GP stage to match the program status of the host with the donor. In this example, the host only defines the variable `array` to be available at the implantation point (line 2 in Figure 5.9a). This variable appears in Listing 5.1 as the first line: `var array_GLOBAL_DECL : [Int] = []`. We use a global variable to be able to connect the program states at any point in the code of the organ. For example, we might want to match it inside an internal function.

Figure 5.11 shows the inputs of the GP process that we construct in the organ extraction stage. The first input is the array of statements in the over-organ (Figure 5.11a). GP will try in different individuals to either include or exclude statements. Initially, we start with all the statements excluded (value `NO` in Figure 5.11a). The second input of the GP stage is the holes mapping space in Figure 5.11b. Note that each of the hole has type that we use to restrict the potential mappings. In this simple example, we can always match Java `int` type to Swift `Int` type, but in general, τSCALPEL will also search over the space of potential type mappings that we detail in Section 5.4. The purpose of the variable mapping is to connect the program status of H to the one of the over organ. A value of `None` here means that the variable keeps its initial declaration in the host.

Organ reduction and adaption the second stage of τSCALPEL uses genetic programming (GP) to transplant, reduce, and adapt the organ. The inputs of GP are the results of the static analysis stage, that we list in Figure 5.11 and that we discussed above. τSCALPEL also requires the organ test suite for specifying the semantics of the feature that guides the GP process. Different organ test suites lead to different features to be transplanted. The test cases in Figure 5.10 captures the current desired feature in this running example: computing the sum across *all* the elements in the input array. Note that in the initial code, the donor was computing the sum only for the positive elements in the input array.

To implement this functionality, GP must not transplant the code that turns the negative elements in the input array into 0. One can imagine different functionalities that might be obtained from this code such as computing the sum across all the *positive* numbers of the array that would also require the complete code of the function `processArray`.

Section 5.4.3 details our GP approach. At a high level, τSCALPEL's GP searches on the holes mapping space, type mapping space, and statements array, by trying different lines to be included in the organ and different type and hole mappings, until it finds an individual to pass all the test cases in the input test suite.

The search for an individual to pass the entire test suite is guided by our fitness function (Section 5.4.4) that will allow GP to select the individuals that are closer to passing the entire test suite. In this fitness function we evaluate the execution of an individual under the input test suite. For example, an individual that passes one of the test cases is better than an individual that does not pass any. GP uses mutation and crossover operators, similar with the case of monolingual software transplantation.

To assess how close an individual is to passing the entire test suite, τSCALPEL uses a phasic fitness function that Section 5.4.4 describes in detail. At a high level, our fitness function first checks the compilation of an individual and considers the compilation errors to give a higher score to an individual that is closer to compile than to an individual that is further away from a successful compilation. Once an individual compiles, our fitness function will rank higher an individual with less compilation errors than one with more. Next, we actually execute the test suite. For each test case we give more fitness points to an individual that executes the test case without errors; and finally, the maximum fitness value (for a test case) for an individual that produces the correct output for the considered test case. We sum the fitness values that a individual obtain across all the test cases and the other fitness components mentioned before to obtain the fitness of an individual. This fitness values rank the individuals in the GP process.

The performance of our GP algorithm (*i.e.* execution time) is correlated with the size of the search space. The size of the search space is defined by: the number of core APIs that the organ in the donor language (Java in this case); the number of core APIs available in the language of the host (Swift in this case) that our APIExtractor identifies; the size of the over-organ; and the number of

```
run1=`.build/debug/MyPackage 3 5 7`
run1Output='15'
run2=$(.build/debug/MyPackage 1)
run2Output='1'
run3=$(.build/debug/MyPackage 3 7 -11)
run3Output='10'

if [[ "$run1" == *"$run1Output"* ]]
    then echo "PASS"
    else echo "FAIL"
fi

if [[ "$run2" == *"$run2Output"* ]]
    then echo "PASS"
    else echo "FAIL"
fi

if [[ "$run3" == *"$run3Output"* ]]
    then echo "PASS"
    else echo "FAIL"
fi
```

Figure 5.10: Multilingual transplantation: the acceptance test of the transplant. For this example transplant we provided the test suite as a bash script that checks the output. These 3 test cases define the functionality to be transplanted as computing the sum of all the positive numbers in the array.

variables that are available at the implantation point in the host that GP matches to variables in the organ.

The chromosome of the successful individual is shown in Figure 5.12. In this case GP all the statements in the original code of D, besides line 14 in Figure 5.9b that was making 0 the negative elements. Other possible alternatives would be to don't call at all the function `processArray`.

Finally, Figure 5.13 shows the chromosome of the successful individual instantiated in Swift code. This is the final result of the transplantation that is to be implanted into the host at the implantation point.

Organ implantation: is the final stage of multilingual software transplantation. This is the simplest of the stages and it simply involves adding a call to the organ interface at the implantation point. The parameters of this call are all the variables in scope at the implantation point. Figure 5.14 shows the host beneficiary. The line that the organ implantation stage of MultiST added is at line 2.

5.4.6 Limitations

The capabilities of τSCALPEL to translate and transplant an organ, from a donor to a host that are written in different programming languages are limited first by the completeness of our translation rules, and secondly by the speed of the organ synthesis approaches. In τSCALPEL we implemented only a subset of these translation rules, that allowed τSCALPEL to successfully execute the experiments in our benchmark that consists of non-trivial real world programs.

Since the number of translation rules between two different languages can be potentially much bigger that we implemented here, there would be a lot of cases where the current implementation

```
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_4 = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_3 = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_6 = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_5 = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_2 = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_1 = NO;
STM_IF_THEN_ELSE_STATEMENT_0 = NO;
STM_WHILE_STATEMENT_2 = NO;
STM_WHILE_STATEMENT_1 = NO;
```

(a) The array of statements in the over organ that τSCALPEL uses to decide which statements to keep in the organ.

```
Var_Name_a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG :
    myboxaArray_OF_Java_PRIMITIVE_int → None

Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG :
    Array_OF_Java_PRIMITIVE_int → None

Var_Name_sum_ClassName_donor_FunctionName_computeSum_2 : Java_PRIMITIVE_int →
    array_GLOBAL_DECL : [Int], None

Var_Name_i_ClassName_donor_FunctionName_computeSum_2 : Java_PRIMITIVE_int →
    array_GLOBAL_DECL : [Int], None

Var_Name_i_ClassName_donor_FunctionName_processArray_2 : Java_PRIMITIVE_int →
    array_GLOBAL_DECL : [Int], None


Var_Name_args_ClassName_donor_FunctionName_mainFunction_FORMAL_ARG :
    Array_OF_Java_PRIMITIVE_int → array_GLOBAL_DECL : [Int], None

Var_Name_d_ClassName_donor_FunctionName_mainFunction_2 : Java_OBJECT_donor →
    array_GLOBAL_DECL : [Int], None

Var_Name_sum_ClassName_donor_FunctionName_mainFunction_2 : Java_PRIMITIVE_int →
    array_GLOBAL_DECL : [Int], None
```

(b) The possible mapping space between host and donor. τSCALPEL will select for each of the variables in the organ one of the possible mappings. A mapping to None means that the variable keeps it's original declaration from the Donor.

Figure 5.11: The inputs of the GP stage.

will fail to tackle. For example, we do not consider in our translation rules java streams. When such construct are reached in a multilingual transplantation process, we could potentially augment τSCALPEL's implementation with new translation rules. The time it takes our GP based organ synthesis process is related to the size of the organ. Our organs' sizes are not trivial (Section 3.3), but still, for much bigger sizes, the time required for the organ transplantation process might not be feasible. To tackle these cases we could potentially parallelize our implementation on multiple machines, and / or augment our rule based translation system that will reduce the search space for our search.

Currently, τSCALPEL translates and transplants between same programming paradigms. We believe that in principle, τSCALPEL could also translate and transplant between different paradigms languages, such as from Java into C. However, in those cases, we believe that the complexity of the

```
Lines:
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_4  = NO;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_3  = YES;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_6  = YES;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_5  = YES;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_2  = YES;
STM_STATEMENT_WITHOUT_TRAILING_SUBSTATEMENT_1  = YES;
STM_IF_THEN_ELSE_STATEMENT_0  = YES;
STM_WHILE_STATEMENT_2  = YES;
STM_WHILE_STATEMENT_1  = YES;


Mappings:
Var_Name_a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG  :
    myboxaArray_OF_Java_PRIMITIVE_int  → None

Var_Name_a_ClassName_donor_FunctionName_processArray_FORMAL_ARG  :  Array_OF_Java_PRIMITIVE_int
    → None

Var_Name_sum_ClassName_donor_FunctionName_computeSum_2  :  Java_PRIMITIVE_int  →
    array_GLOBAL_DECL : None

Var_Name_i_ClassName_donor_FunctionName_computeSum_2  :  Java_PRIMITIVE_int  →
    array_GLOBAL_DECL : None

Var_Name_i_ClassName_donor_FunctionName_processArray_2  :  Java_PRIMITIVE_int  →
    array_GLOBAL_DECL : None


Var_Name_args_ClassName_donor_FunctionName_mainFunction_FORMAL_ARG  :
    Array_OF_Java_PRIMITIVE_int  → array_GLOBAL_DECL : [Int]

Var_Name_d_ClassName_donor_FunctionName_mainFunction_2  :  Java_OBJECT_donor  →
    array_GLOBAL_DECL : None

Var_Name_sum_ClassName_donor_FunctionName_mainFunction_2  :  Java_PRIMITIVE_int  →
    array_GLOBAL_DECL None


FITNESS: [ 1.00 ]
```

Figure 5.12: The chromosome of the successful individual. The rest of mappings have no influence in this individual, since there are no statements involving them. RETURN_MARKER appears in all the individuals, as this is the organ entry point.

translation rules will increase since them will have to incorporate transforming classes into similar C structs. The effort for constructing and maintaining such rules might be too big in such case.

5.5 Experiments

In this section, we present a comprehensive empirical study to assess the efficiency and the effectiveness, modulo testing, of τSCALPEL at transplanting functionality (organs) from real world programs. In all our experiments, the host and the donor programs are written in different languages: τSCALPEL transplants ten different organs from ten different donors into ten different hosts between two different language pairs: five transplants are from Java donors into Swift hosts; the other five transplants are from Fortran donors into Python hosts.

```
var array_GLOBAL_DECL : [Int] = []

class TRANSPLANTED_CLASS_donor {

    func TRANSPLANTED_FUNCTION_computeSum (
         a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG : inout [Int]) ->Int

    {
        var sum_ClassName_donor_FunctionName_computeSum_2 : Int
        sum_ClassName_donor_FunctionName_computeSum_2 = 0
        var i_ClassName_donor_FunctionName_computeSum_2 : Int = 0
        while i_ClassName_donor_FunctionName_computeSum_2 <
             a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG .count {
            sum_ClassName_donor_FunctionName_computeSum_2 =
                 sum_ClassName_donor_FunctionName_computeSum_2 +
                 a_ClassName_donor_FunctionName_computeSum_FORMAL_ARG [
                 i_ClassName_donor_FunctionName_computeSum_2 ]
            i_ClassName_donor_FunctionName_computeSum_2 += 1
        }
        return sum_ClassName_donor_FunctionName_computeSum_2
    }

    func TRANSPLANTED_FUNCTION_processArray (
         a_ClassName_donor_FunctionName_processArray_FORMAL_ARG : inout [Int])
    {
        var i_ClassName_donor_FunctionName_processArray_2 : Int = 0
        while i_ClassName_donor_FunctionName_processArray_2 <
             a_ClassName_donor_FunctionName_processArray_FORMAL_ARG .count {
            if( a_ClassName_donor_FunctionName_processArray_FORMAL_ARG [
                 i_ClassName_donor_FunctionName_processArray_2 ] < 0 )
            {
            }
            i_ClassName_donor_FunctionName_processArray_2 += 1
            }
    }

}

func organInterface(array_GLOBAL_DECL_param: [Int]) -> Int
{
    array_GLOBAL_DECL = array_GLOBAL_DECL_param
    var d_ClassName_donor_FunctionName_mainFunction_2 : TRANSPLANTED_CLASS_donor =
         TRANSPLANTED_CLASS_donor ()
    d_ClassName_donor_FunctionName_mainFunction_2
         .TRANSPLANTED_FUNCTION_processArray(a_ClassName_donor_FunctionName_processArray_
         FORMAL_ARG : &array_GLOBAL_DECL )
    var sum_ClassName_donor_FunctionName_mainFunction_2 : Int =
         d_ClassName_donor_FunctionName_mainFunction_2
         .TRANSPLANTED_FUNCTION_computeSum(a_ClassName_donor_FunctionName_computeSum_
         FORMAL_ARG : &array_GLOBAL_DECL )
    return sum_ClassName_donor_FunctionName_mainFunction_2
}
```

Figure 5.13: The successful individual instantiated.

We organise our study around four research questions, designed to assess τSCALPEL effectiveness and efficiency. The first three research questions are about effectiveness; the last research question is about efficiency. We report a transplant as successful in our result tables in the columns *PR* (Passing Rate). To be successful, a transplanted organ needs to pass **all** the test cases in a test suite.

```
 1  func compute(array : [Int]) -> Int {
 2      return organInterface(array_GLOBAL_DECL_param: array)
 3  }
 4
 5  var input = [Int]()
 6
 7  for arg in CommandLine.arguments {
 8      if(Int(arg) != nil){
 9          input.append(Int(arg)!)
10      }
11  }
12
13  print(compute(array : input))
14
```

Figure 5.14: The successful individual called in the host at the implantation point (line 2) in the host beneficiary system.

Table 5.1: The Java to Swift corpus.

Project	Type	Size	Organ
Compress [74]	Donor	24k	LZF compression
TSWeChat [84]	Host	12k	-
jGnash [144]	Donor	92k	Currency conversion
SIP-Calculator [83]	Host	0.5k	-
NewsReader [77]	Donor	1.2k	News filter
The Oakland Post(*TOP*) [85]	Host	15k	-
OWASP [79]	Donor	65k	XSS vulnerability checker
Firefox-iOS [102]	Host	420k	-
Sorter [296]	Donor	0.5k	Fast sorting algorithm
AppLove [16]	Host	10k	-

Table 5.2: The Fortran to Python corpus.

Project	Type	Size	Organ
clawpack [69]	Donor	106k	DASKR Solver
PyDAS [80]	Host	28k	-
FLAP [76]	Donor	4k	CLI Parser
dropship [93]	Host	0.3k	-
fortranlib [104]	Donor	16k	`quadratic_reduced_sp`
veusz [323]	Host	48k	-
fson [107]	Donor	1.7k	JSON parser
ansible-xml(*ansible*) [15]	Host	1.3k	-
Slycot [82]	Donor	82k	Lyapunov equation solver
pySchrodinger(*pySchro.*) [255]	Host	0.2k	-

Corpus: To build our Java to Swift corpus, we used an online list of open source iOS applications[7]. This list groups applications by domain. In each domain, in the order they appeared in the list, we selected, as our host, the most popular Swift application. This systematic selection process selected the following domains: *Browser*, *Calculator*, *Communication*, *Developer*, and *News*. One of the authors examined these applications to identify missing functionality, then mined GitHub [109] for Java applications that implemented that functionality. Table 5.1 shows the resulting corpus for Java to Swift transplantations. For the Fortran to Python transplantation experiments unfortunately we did not have such a list of popular host programs. In this case we uniformly picked five programs from

[7]We used the list at `https://github.com/dkhamsing/open-source-ios-apps`, visited 2016-01-12.

the first top ten results on GitHub for the query `language: FORTRAN`, to identify popular Fortran organs. These systematically selected programs are our donors in Table 5.2. As above, one author identified transplantable functionality in each donor, then manually mined GitHub for suitable hosts written in Python. Table 5.2 shows the subjects for Fortran to Python transplantations.

All the programs in our corpus are nontrivial, real world programs from various problem domains. A diverse set of Android apps dominate our Java programs, while our Fortran programs are all numerical. We picked Java to Swift and Fortran to Python, because developers are already moving functionality between these languages, sometimes from scratch and sometimes via manual multilingual transplantation. Android app developers who wish to expand into the iOS market drive the interest in Java to Swift. Fortran remains the preeminent simulation and modeling language in the physical sciences, although it is increasingly facing competition. Especially for prototypes, many scientists are turning to Python[8]. MultiST promises to ease this transition by speeding the reuse of Fortran code in Python projects.

In Table 5.1, `Compress` is a Java library for LZF [339] compression; `TSWeChat` is an iOS chat app that supports sending files. We transplant `Compress`'s compression functionality into `TSWeChat` to compress outbound files. As discussed in Section 4.2, `SIP-Calculator` calculates the future value of Systematic Investment Plan (SIP) payments, but does not support currency conversion. `jGnash`, is a popular "personal finance manager with many of the same features as commercially-available software" [144]. We transplant its currency conversion functionality into `SIP-Calculator`. `The Oakland Post` is an iOS app for the newspaper of the same name. `NewsReader` is an Android application that for browsing and filtering news. We transplanted `NewsReader`'s filtering functionality into `The Oakland Post`. Next, we transplanted an HTML sanitizer organ that checks HTML for XSS attacks from `java-html-sanitizer` into `Firefox-iOS`. The Apple Store restricts users to viewing only reviews for their region. `AppLove` bypasses this restriction. Currently, `AppLove`'s reviews are unsorted. `Sorter` is an efficent, multithreaded command line tool for sorting strings. To sort reviews, we transplant `Sorter` into `AppLove`.

In Table 5.2, we transplanted the DASKR Solver organ from `clawpack`, into `PyDAS`, a differential algebraic system solver. We transplanted the CLI Parser organ from the `FLAP` donor, into the host `dropship`, a utility that allows users to instantly transfer files between Dropbox accounts by using only their hashes. Next, we transplanted the `quadratic_reduced_sp` computation organ from `fortranlib`, to `veusz`. `fortranlib` is a collection of scientific functions. `veusz` is a scientific python plotting application. We transplanted the JSON parsing organ from `fson`, into `ansible-xml` host, an ansible module that allows the manipulation of bits and pieces of XML files and strings. `pySchrodinger` is a python solver for 1D Schrodinger equation. For `pySchrodinger`, we transplanted a Lyapunov equation solver from `Slycot`.

[8]At University College London, programmers who help scientists write and maintain their experimental code anecdotally report increasing interest in porting code from Fortran to Python.

Table 5.3: Transplantation results over 20 runs. The “Unanimously Passing runs (PR)” column shows the number of runs that passed all test cases in all test suites. Each “All” column reports coverage of the entire host for each test case; each “Org%” column reports the coverage of the transplanted organ.

Java → Swift

		Unanimously	Test Suites									Time (min)	
Donor	Host	**Passing**	Regression			Regression++			Acceptance			Avg.	Std.
(Java)	*(Swift)*	**Runs (PR)**	**PR**	All%	Org%	**PR**	All%	Org%	**PR**	All%	Org%		Dev.
jGnash	SIP - Calculator (DONE)	**13**	20	15.7	82.4	20	28.8	82.5	13	7.2	96.7	397	127
Compress	TSWeChat (DONE)	**16**	20	61.4	52.7	20	61.4	52.8	16	37.5	72.9	428	243
Sorter	AppLove (DONE)	**20**	20	0.0	0.0	20	18.2	96.1	20	18.2	96.1	7	3
OWASP	Firefox-iOS	**14**	20	43.0	0.0	20	43.0	73.2	14	9.2	79.3	236	73
NewsReader	The Oakland Post	**19**	20	0.0	0.0	20	23.3	84.1	19	11.7	87.9	2	1
	Average	**16**	20	24.0	27.0	20	34.9	77.7	16	16.8	86.6	214	89
	Total / %	**82**	100	-	-	100	-	-	82	-	-	71(h)	447

Fortran → Python

		Unanimously	Test Suites									Time (min)	
Donor	Host	**Passing**	Regression			Regression++			Acceptance			Avg.	Std.
(Fortran)	*(Python)*	**Runs (PR)**	**PR**	All%	Org%	**PR**	All%	Org%	**PR**	All%	Org%		Dev.
fortranlib	veusz	**11**	20	0.0	0.0	18	7.8	100	11	3.2	100	3	2
fson	ansible-xm	**17**	19	76.9	22.2	18	76.9	74.2	18	31.2	88.1	27	23
Slycot	pySchrodinger	**8**	20	0.0	0.0	20	97.6	82.1	8	97.6	87.9	557	349
FLAP	dropship	**15**	20	0.0	0.0	18	47.9	100	16	47.9	100	238	83
PyDAS	clawpack	**12**	20	37.3	0.0	20	37.3	21.4	12	18.8	57.7	103	13
	Average	**12**	19	22.8	4.4	18	53.5	75.5	13	39.7	86.7	185	94
	Total / %	**63**	99	-	-	94	-	-	65	-	-	61(h)	470

Experimental Setup: We randomly set parameters to create the initial population. For the “holes” chromosome, we convert the scores of each candidate for filling a hole at the implemenation point (Section 5.4.3) into a probability distribution; for the “statements” chromosome, we uniformly randomly select statement from the over-organ. τSCALPEL applies its crossover operator with a probability of 0.5. It mutates an individual with a probability of 0.5, selecting which mutation operator uniformly. Our solution is based on genetic programming which is stochastic by nature. Because of this, we run all our experiments 20 times to provide a set of results that are robust in the presence of such variability.

To validate the results of the transplants that τSCALPEL did we used three different test suites that were completely hidden from τSCALPEL. In the GP process, τSCALPEL uses the organ test suite, while for the evaluation of the transplant in the rest of this section we used three different test suites to be sure that the evaluation is independent from the implementation of τSCALPEL and to be sure that our transplants do not overfit the test suite that is used in the GP process. That is, the organ test suite is used only by τSCALPEL in the GP process, while the evaluation test suites are used just for the evaluation and are invisible for τSCALPEL.

5.5.1 Is τSCALPEL Disruptive?

τSCALPEL transplants alien code between two different programs written in different programming languages. Thus, it is important to check whether the transplant disrupts any of the pre-existing functionality of the host that we seek to preserve. For example, the JSON parser organ that we extracted from `fson` had a side effect that modified its input string. This caused one of the `ansible-xml` regression test to fail (Table 5.3, `fson` to `ansible-xm` transplant). **RQ1** therefore asks whether τSCALPEL can transplant code without breaking the initial functionality of the host system. **RQ1:** *"How many organs manage to leave the pre-existing functionality of the host undisturbed?"*

We first answer **RQ1** with existing test suites, those available with our hosts and designed by their developers. Table 5.3 reports the results in the Regression **PR** (passing run) column: 100 transplants out of 100 passed the initial regression test suite available with the host system for the Java to Swift transplants. For Fortran to Python transplants, 99 out of 100 transplants passed the initial regression test suites. Lack of coverage undermines this initial result. The All% columns report the coverage of each test suite for the entire host; the Org% columns report their coverage for the organ, the code that we transplanted. As we can see in the 'Regression' columns, the coverage achieved is insufficient: most organs are not even covered.

To more thoroughly evaluate our transplants and to provide more evidences for the fact that our transplants didn't introduced regression faults, we took steps to improve the test adequacy of the test suites. Specifically, one of the authors manually designed additional test cases with the aim of executing the organ more thoroughly. It is expected that in some cases the organ might break some existing functionality because of its side effects. To identify those cases, we try to leverage the exiting test suite. When the existing test suite is not achieving high coverage, τTrans requires additional regression test cases to be written specially for the organ. We believe that writing the test cases as a specification for the organ is in general easier than implementing the organ itself.

Table 5.3 reports the results of these manually augmented test suites in its *Regression++* columns. Here, **PR** is unchanged. The Regression++ test suites achieved considerably higher coverage, especially over organs, as the "Regression++" columns show. Our augmented regression test suites caught 6 regression bugs that the initial regression test suites missed. In sum, we answer **RQ1** with:

Finding 1: τSCALPEL transplants and translates functionality that passes all regression tests in 100% of the cases for the Java to Swift transplants and in 99% of the cases for the Fortran to Python transplants; and augmented regression tests in 100% of the cases for the Java to Swift transplants and in 94% of the cases for the Fortran to Python transplants.

This finding shows that we can transplant code, written in a different language, into a host system and still pass its initial regression test suite and an augmented one specially designed to reveal regression faults in the transplanted organ's code.

5.5.2 Does τSCALPEL Move Functionality?

Merely transplanting code that does not break the initial functionality is insufficient for a transplant to be successful: the postoperative host should also have the transplanted functionality. This observation motivates us to ask **RQ2:** *"How many multilingual transplantations augment the functionality of the hosts?"*

We use acceptance tests to answer **RQ2**. Since the initial hosts lack the functionality that we transplant, we need an acceptance test suite for each scenario. While we believe this acceptance test suite generation problem is amenable to automation, for the purpose of implementing an overall research prototype for the present paper, we performed this step manually. Specifically, we manually created and added tests until we satisfactorily covered the organ (*i.e.* achieved more than 70%) or when further coverage improvement required domain expertise that we lack. For the DASKR Solver organ, for example, further increasing its coverage requires crafting complex differential equations, a step we omitted.

Table 5.3 answers **RQ2** in its "Acceptance" columns. For Java to Swift transplants, 82 (out of 100) organs pass acceptance testing; for Fortran to Python, 65 organs (out of 100 organs) do so. The columns "All%", and "Org%" under "Acceptance" show that the average coverage on the entire postoperative host system is 16.78% and 39.75% for Java to Swift and Fortran to Python respectively. When we consider only the organs, coverage jumps to 86.62% for the Java to Swift transplants and 86.73% for the Fortran to Python transplants. In all but one case, we were able to build high coverage (70%) Regression++ test suites, despite our lack of domain knowledge, that achieved coverage greater than 70% over the organ. The answer to **RQ2** is that 147 transplants out of 200 pass the acceptance test suites:

Finding 2: τSCALPEL transplants and translates functionality that passes all acceptance tests in 82% of the cases for the Java to Swift and in 65% for the Fortran to Python transplants.

5.5.3 Success Rate

A successful transplant simultaneously leaves all pre-existing functionality undisturbed, while also augmenting the host with the new functionality: we therefore consider a transplant to be "successful" if and only if it passes all regression and acceptance tests. **RQ3** intersects **RQ1** and **RQ2** and asks *"How many transplants pass all the test cases in all the regression and acceptance test suites?"*

We say that a transplant that passes all regression and acceptance tests, does so "unanimously". Table 5.3's "Unanimously Passing Runs (PR)" column records how often this occurred in our experiments and answers **RQ3**. On average, 16.4 transplants out of 20 are successful for Java to Swift transplantations; for Fortran to Python transplantations, 12.6 out of 20 unanimously pass. The answer to **RQ3** is promising, providing evidence that MultiST is feasibility:

Finding 3: 82% of the Java to Swift and 63% of the Fortran to Python transplants are successful: they do not disrupt the host's pre-existing functionality, while augmenting host with the transplanted functionality.

Defining success as as unanimous passing, as we do, means that these results establish a lower bound on the utility of MultiST because they do not account for the value of a partially successful transplantation, which may, even if it had a low pass rate, be easier for a developer to adapt than manually performing the transplantation from scratch.

5.5.4 Computational Cost

Finally, we assess the efficiency of τSCALPEL, asking **RQ4:** *"What is the computational cost of τSCALPEL?"*. A successful transplant is not enough; it must be achieved quickly enough to be acceptable in practice. To answer this question, we report the elapsed time τSCALPEL used.

Table 5.3 answers **RQ4** with its "Time" columns on the far right. Over 20 runs measured with GNU Time, these columns report the average runtime and standard deviation. Our answer to **RQ** is

Finding 4: Over our corpus, τSCALPEL averages 214 minutes to transplant from Java donors into Swift hosts and 185 minutes to transplant from Fortran donors into Python hosts.

We believe these runtimes are reasonable since τSCALPEL replaces human effort. Given its inputs, τSCALPEL automatically translates and transplants functionality. For τSCALPEL, these inputs are the organ entry point in the donor, the implantation point in the host, and an organ test suite. Given these inputs, τSCALPEL translates and transplants the required functionality without further human intervention. Compare this to manual transplantation. The software engineer must identify the required functionality and extract it from the donor, copy it into the host, and connect the free variables in the organ to those in scope in the host at the implantation point or add adaptor code to initialize them.

None of these steps are trivial. Identifying functionality in the donor must start with identifying the start of that functionality, *i.e.* the organ entry point. Then the developer must identify all the code that might be called from the organ's entry point (*i.e.* organ), *viz.* she must manually slice the donor. The path from a program's entry point to the organ's entry point may contain adaptor code that the organ will need in the host (*e.g.* initialization routines or data structure construction). Here again, the developer must resort to manual slicing. When copying the organ into the host, the developer must juggle translating the code with searching for bindings for the organ's free variables and handling namespace conflicts. Finally, the developer must validate their translation and transplantation. Good practice here dictates testing, which entails augmenting the host's test suite, with new test cases that exercise the new functionality.

In short, manual multilingual transplantation subsumes τSCALPEL's inputs: when manually performing multilingual software transplantation, developers generate each input that τSCALPEL requires. They must identify the organ entry point, although not explicitly annotating it. However the annotation is trivial: the user of τSCALPEL only need to add a comment in the source code of the Donor program. Next, developers must identify the implantation point in the host. Finally, they must validate the organ in the postoperative host. If they resort to testing here, they must generate an organ test suite. τSCALPEL eliminates the rest of the manual work, freeing the developer to work on other tasks.

Even if an average software engineer could potentially follow a similar approach with τSCALPEL and manually translate and transplant an organ in a relatively short time, the fact that all the programs are from different problem domains and also written across 4 different language means that probably the engineer would not be familiar with all of them. For the unfamiliar problem domains / languages, the engineer might need more time. Considering that τSCALPEL automate this process, it allows the software engineer to concentrate on more interesting and challenging problems than translating and transplanting the code from a reference implementation.

For the above reasons, we believe multilingual software transplantation to be useful even in cases where the transplant fails to pass all the validation test cases. In those cases, the human software engineer has a starting point for the implementation of the organ, where (s)he can fix the falling test cases. We believe this is easier than implementing the organ from scratch.

5.5.5 Do Existing Translators Achieve Transplantation?

The existence of automated translation systems raises the question of whether automated translation tools could achieve multilingual transplantation. Unfortunately, we found that neither of the existing translation approaches can be employed for automated transplantation, without considerable further work. We therefore suggest the further development of hybrid translation and transplantation techniques, beyond τSCALPEL, as a topic for future work. In the remainder of this section, we report our attempts to adopt and re-purpose existing translation techniques for the problem of transplantation.

We enabled τSCALPEL to output the organ, both in the language of the donor and in the language of the host. With these two versions of the organ, we can now use existing translators on the organ in the language of the donor to obtain the organ in the language of the host (for our Java organs).

We searched on Google for "java to swift translator" on 20.10.2018 and looked at the first five pages of the results. Here we found only two existing rule-based translators: `patniemeyer/j2swift`[9] and `eyob/j2swift`[10]. Unfortunately, we found that they produce uncompilable code; they do not translate API calls so the Swift code that they produce calls Java APIs. To handle APIs, rule-based translators need rules for all relevant APIs across the donor's language and the host's language. Maintaining these rules as each languages' ecosystem evolves, which can sometimes be rapid, can be prohibitively expensive. Thus, it is not surprising that we were not able to find any rule-based translators for the language pairs in our experiments that handle API translations. In contrast, τSCALPEL handles APIs by design, because we use rules only for the core language constructs and GP for the rest, including APIs. In short, this problem of translating APIs is a challenge for current formulations of rule-based translation when applied to transplantation.

To compare τSCALPEL with SMT approaches, we considered Nguyen *et al.*'s SemSMT [232], an SMT translator from Java into C#, and Green *et al.*'s Phrasal [113], a general toolkit for SMT, not tailored for code, that is available on Github[11]. Unfortunately, SemSMT is not available either on Github [12] or on its webpage[13]. For Phrasal, we were unable to find an existing Java to Swift corpus

[9] `https://github.com/patniemeyer/j2swift`
[10] `https://github.com/eyob--/j2swift`
[11] `https://github.com/stanfordnlp/phrasal`
[12] `https://github.com/SoftwareEngineeringToolDemos/ICSE-2014-semSMT`
[13] `http://home.engineering.iastate.edu/~anhnt/Research/StaTran/?page=introduction`

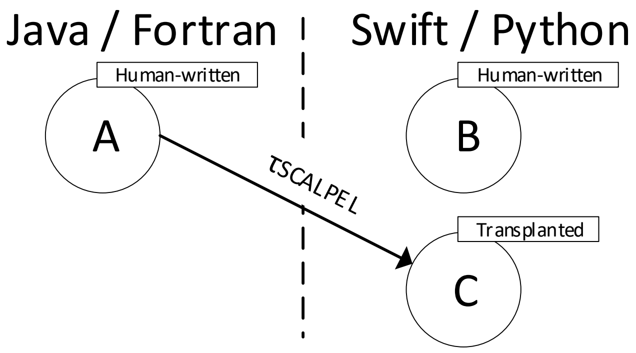


Figure 5.15: Case study template: each circle implements the same functionality.

for training, so we manually generated a aligned Java to Swift training set with 974 LOCs. We then applied Phrasal, trained on this data, to the organ translations in our five Java to Swift experiments (Section 5.5). Unfortunately, all of these translation attempts failed. We suspect the small training set is the reason; we discuss the challenge of training data next.

5.5.6 Threats to Validity

As is commonly observed in empirical software engineering studies such as ours, our evaluation set may not be fully representative. Our subject selection process, detailed in **Corpus** (Section 5.5), mitigates this external validity threat. Weak proxies are a threat to internal validity, and this has affected some previous GI work [257]. To mitigate this threat, we manually examined each test case to confirm that it checked the output of the program. In fact, we leverage weak proxies as a subfitness function (Section 5.4.4). The passing rate, code coverage and runtime are not necessary a good measure of the utility of transplantation. We address this threat to construct validity in Section 5.6 for all our 10 transplants.

5.6 Case Studies

Section 3.3 shows that τSCALPEL can transplant code without breaking the pre-existing functionality of the host system, add the functionality that previously existed only in the donor system, and that the transplantation process is sufficiently efficient in terms of execution time. But are the transplants that our technique produces useful and correct? Furthermore, how confident are we in our testing? We approximated the correctness of our transplants with respect to the augmented test suite that we manually generated, as explained in the previous section. A manual augmentation process may increase confidence in transplant correctness and acceptability, but any such manual test augmentation might introduce bias in assessing the correctness of the organ.

To address the twin questions of the usefulness and potentially incorrect behaviour of our τSCALPEL, we conduct ten case studies, one for each of the transplanted organ.

Each case study seeks to compare human-written functionality against the similar functionality translated and transplanted by τSCALPEL. Figure 5.15 depicts this relationship. We mined GitHub

Oracle	Donor	Experiment
Swift Oracles		
swift-sorts[304]	Sorter	Identical output on 10,000 uniformly random generated arrays.
ExchangeRateApp [75]	jgnash	Identical output on 10,000 uniformly random generated values.
NSData-Compress [78]	compress	Different compressed files, but identical decompression on 100 files, uniformly picked from our corpus.
NewsApp [230]	news reader	Identical filtering results for the 50 top news as appeared on Google on 24-01-2020 .
SwiftSoup [306]	OWASP	Identical sanitize results on a set of 50 XSS.
Python Oracles		
Clawpack [69]	Clawpack	Identical results on a set of 5 differential equations.
argparse [19]	FLAP	Identical results on 5 test cases that one of the authors wrote.
EquationSolver [305]	fortranlib	Identical results on 10 quadratic equations that one of the authors wrote.
json [97]	fson	Identical results on 17 text cases that we found in the JSON standards.
scipy [281]	Slycot	Identical results on a set of 100 automatically generated uniformly random test cases.

Table 5.4: Our case study results: the column ‘Oracle’ names a human-written program that implements the functionality.

to find matches for node **B** in Figure 5.15 by issuing searches using keywords related to the organs' core functionality and restricted to the host language. We sought human-written organs for two reasons: 1) their existence shows that human developers care about implementing the functionality that we transplanted automatically thereby providing evidence for usefulness of τSCALPEL; and 2) the human-written organs provide us with opportunities for more thorough and unbiased testing using them as oracles for the organs that we transplanted. We use them to answer **RQ5**: *“How do our transplanted organs compare with human-written similar functionalities, in the language of the host?”*

To remove potential bias that might otherwise arise in our manual augmentation of the host's regression and acceptance tests, we use random testing wherever the organ's core functionality allowed this. This is possible because we can use the alternative human-written systems as oracles. We describe this process for each of the case studies below.

Sorting Organ: The first case study investigates the *Sorting* organ that we transplanted from `Sorter` into `AppLove`. The sorting organ sorts the reviews from `AppLove` in lexicographic order. We searched GitHub for Swift programs that contain “sort” in their names and found the program `swift-sorts` (line 1 in Table 5.4).

To test our transplant using `swift-sorts` as oracle, we uniformly randomly generated $10,000$ string arrays. First, we randomly selected a length l, between 0 and $1,000$. Next, we uniformly randomly generated l strings. For each string we uniformly randomly picked its length, and then uniformly randomly generated its characters. Finally, we ran both the sorting organ and the sorting functionality of `swift-sorts` and compared the outputs that they produced. In all cases, the outputs were identical, providing further evidence for the (sufficiently) correct behaviour of our transplant.

Currency Exchange Organ: The second case study investigates the *Currency Exchange* organ that we transplanted from `jGnash` into `SIP-Calculator`. This organ converts between different currencies. We searched GitHub for Swift programs that contain "exchange" in their names and found `ExchangeRateApp` (Table 5.4). We uniformly randomly generated $10,000$ float values and $10,000$ initial currency $\rightarrow$ target currency pairs. Finally, we executed both the organ and the exchange functionality from `ExchangeRateApp` on the generated inputs. In all the cases, the outputs were identical.

Compress Organ: The third case study investigates the *Compress* organ that we transplanted from `Compress` into `TSWeChat`. This organ compresses the files prior to *TSWeChat* sending them. We searched GitHub for Swift programs that contain "compress" in their name and found `NSData-Compression` (line 3 in Table 5.4).

In this case, the compressed outputs of the *Compress* organ differs from the compressed output of `NSData-Compression`, but they decompress to the same files. They are semantically equivalent in this sense. To address such potential differences, we use metamorphic testing [67]. We uniformly randomly picked 100 different source code files from our experimental corpus (*i.e.* the project folders of all the programs that we used in our empirical study in Section 3.3). Next, we executed both the organ and the compress functionality from `NSData-Compression` on these files. In this case the metamorphic transformation is the decompression. Thus, we used `tar` to decompress the resulting outputs and compare these. Finally, we compared the decompressed files with the original uncompressed ones. In all the cases, the outputs were identical.

News Filter Organ: The fourth case study investigates the *News Filter* organ that we transplanted from `NewsReader` into `The Oakland Post`. We searched GitHub for Swift programs with the keywords "filter news" and found NewsApp [230] (line 4 in Table 5.4).

To identify a valid corpus for comparing the organ and the human written functionality, one of the authors manually selected the top 50 news from Google news on 24-01-2020. Next, we defined 5 filtering out categories and provided these inputs to both of the organs. Both of the organs filtered out exactly the same 37 news. We explain the identical output as the fact that both the applications were implementing a simple syntax based filtering strategy, where they were filtering out all the articles containing the words to filter for.

XSS Vulnerability Check Organ: The final Swift organ case study is about the *XSS Vulnerability Check Organ* that we transplanted from `OWASP` into `Firefox-iOS`. We searched GitHub for Swit programs with the keywords “XSS Vulnerability” and found SwiftSoup [306] (line 5 in Table 5.4).

To identify a valid corpus for comparing the organ and the human written functionality, we first identified a list of valid XSS attacks at `https://owasp.org/www-community/xss-filter-evasion-cheatsheet`. From this list, we uniformly randomly picked 50 and then run both organs of them, to sanitize the inputs. For both the organs we did not provided a allowed list, but rather relied on the defaults. Finally, we manually inspected the results and observed that in all the 50 cases, both the transplanted and the human written organ, identified the same XSS attacks and sanitized them. There were some syntax differences, but not any semantic ones between the outputs. For example, formatting related differences.

Differential Equation Solver Organ: The case study to investigate a Python organ is about the *Differential Equation Solver* organ that we transplanted from `Clawpack` into `PyDAS`. Fortunately, in this case, `Clawpack` provides a Python interface, that we could call directly from Python to compare it with the transplanted organ.

In this case it was hard to randomly generate differential equations. Thus, one of the authors, manually wrote 5 differential equations and wrote both the human written organ and the transplanted organ on them. The results were identical. In this case, the evaluation was more of a sanity check, as we couldn’t generate a comprehensive set of test cases.

CLI Parser Organ: The next case study is about the *CLI Parser* organ that we transplanted from `FLAP` into `dropship`. `FLAP` is inspired by the python’s argparse [19] and so, we decided to compare it’s implementation in argparse.

Again, since CLI arguments are structured formats, we couldn’t automatically generate random test cases. Here, we produced 5 manual test cases to test the 5 supported CLI formats: short argument name, long argument name, argument with values, flag argument, and nargs. In all the cases, the results of the parsings were the same.

Quadratic Equation Solver: The next case study is about the *Quadratic Equation Solver* organ that we transplanted from `fortranlib` into `veusz`. We automatically generated 100 quadratic equations, by randomizing the values for ‘a‘, ‘b‘, and ‘c‘. In all the 100 cases, both the human written organ and the transplanted one produced the same results.

JSON Parser Organ: The next case study is about the *JSON Parser* organ that we transplanted from `fson` into `ansible-xml`. We though about comparing this organ with the standard implementation in python, the module `json`, by calling the method `json.loads()`.

In this case it was hard to automatically generate random json to test the valid json space, because of the structured format of json. Instead we identified 2 datasets, one from Adobe (`https://opensource.adobe.com/Spry/samples/data_region/JSONDataSetSample.html`) and one from JSON (https://json.org/example.html). In total we have 17 examples that test the different possible JSON constructs, according to the JSON standard. In all the cases, both the human written organ and the transplanted one, parsed the JSONs in the same way.

Lyapunov Equation Solver Organ: Our final case study is about the *Lyapunov Equation Solver* organ that we transplanted from `Slycot` into `pySchrodinger`. We searched GitHub for Python programs with the keywords "Lyapunov Equation" and found scipy [281].

The type signature of the organs take 2 matrices as inputs: 'a' and 'q'. Scipy implements a Lyapunov equation solver it: `scipy.linalg.solve_lyapunov(a, q)`. Next, we uniformly randomly generated lengths between 5 and 20, and values for the elements of the matrices. We produced 100 such inputs and then run both the human written organ and the transplanted organ on them. The results were identical in all the cases.

Table 5.4 summarises our case studies and answers **RQ5**:

Finding 5: Our case studies show that our organs are correct modulo our (uniformly randomly produced) test suites applied to an equivalent "oracle" human implementation that we identified on GitHub for three of our Java to Swift organs.

Since the human developers implemented similar or identical organs to the ones that we transplanted, we also have evidence that τSCALPEL is useful in practice.

When we evaluated the case studies, we also had a look at the performance of the transplanted organs versus the performance of the human written organs. We didn't observed significant differences here: in all the cases, the execution time variations could been attributed to different loads of the operating system, rather than to performance difference of the organs. This might be due to the fact that our organs do not take a significant execution time that is less then 1 second for all the test cases in an individual case study.

The auto-transformed code of the organs follows closely the syntax of the donor code, by preserving the most of the syntax, including the variable names where the names do not collide with the namespace of the host. The translation rules and GP-produced code might indeed be more difficult to read and maintain, particularly when the language pairs come from different codding styles and / or paradigms. For example, in Python, developers often use list comprehension that makes the Python code more readable for them. Since Fortran does not have such a concept, the transplanted code from Fortran into Python will not take advantage of this structure and will be harder to read and maintain for Python developers.

5.7 Transplant Maintainability

An issue that might arise about multilingual software transplantation is about the maintainability of the code transplants and the consequences that it brings for the total cost of ownership. We assume that the donor is a well written program that is readable and maintainable, as otherwise we would select a different donor for transplant. Thus, with our work in software transplantation we aim to obtain an organ with the code as close as possible to the initial donor program to increase the maintainability of the organ and to reduce the total cost of its ownership.

While in the case of monolingual software transplantation (Chapter 4) our approach for transplantation keeps the code of the transplant very similar with the initial code in the donor (besides α-renaming, inlining, and code deletion) this is not the case for multilingual software transplantation.

Since multilingual software transplantation involves code translation, the code of the organ as instantiated into the host system is highly different than the initial code of the donor program. The changes that τSCALPEL does compared with the initial donor code are:

- Use translation rules to translate core language constructs;
- α-renaming to avoid name space conflicts and to map variables in the donor to variables in the host;
- Code deletion to specialize the organ;

If we have a look at our running example's instantiated organ in Figure 5.13 we can see that the code looks quite far from what a human developer would write. In this section we describe some optimisations for future work to make the multilingual organs more readable and maintainable.

The main reason why the code in Figure 5.13 is hard to read is because α-renaming created very long and not very meaningful name. A simple solution for this would be to be sure to compile the organ in a name space that is not used in the host program. Assuming that the names do not collide in the initial donor program (as otherwise the donor program would not be compilable), like this we would not need to do any α-renaming besides the one for mapping the variables in the host to the ones in the organ. Here again we could use the names in the host, which we assume to be readable as they were written by human developers in the host. After this simple optimisation, the code would look like in Figure 5.16. If we compare this with the version of the organ in Figure 5.13 immediately we can see a huge readability improvement.

The second big source of the differences between the organ in the donor and the transplanted organ is because of our rule based translation system that translates the organ from the donor's language into the host's language. In this case there is not any way to keep the code as in the donor as the language of the host is different. Careful designing of the translation rules such that they produce readable and maintainable code is fundamental here to insure a transplant that is close to human written code.

The final source of diverging from the initial organ code in the donor is caused by statement deletions. This shouldn't affect too much the quality of the transplant as it only involves deleting code that is either dead code or cannot function in the context of the host. An optimisation here would

```
var array_GLOBAL_DECL : [Int] = []

class TRANSPLANTED_CLASS_donor {

    func computeSum ( a : inout [Int]) ->Int

    {
        var sum : Int
        sum = 0
        var i : Int = 0
        while i < a .count {
            sum = sum + a [ i ]
            i += 1
        }
        return sum
    }

    func processArray ( a : inout [Int])
    {
        var i : Int = 0
        while i < a .count {
            if( a [ i ] < 0 )
            {
            }
            i += 1
            }
    }

}

func organInterface(array_GLOBAL_DECL_param: [Int]) -> Int
{
    array_GLOBAL_DECL = array_GLOBAL_DECL_param
    var d : TRANSPLANTED_CLASS_donor = TRANSPLANTED_CLASS_donor ()
    d.processArray(a : &array_GLOBAL_DECL )
    var sum : Int = d .computeSum(a : &array_GLOBAL_DECL )
    return sum
}
```

Figure 5.16: The successful individual of our running example instantiated by keeping the original names from the donor and host.

be about deleting empty blocks and using state of the art techniques for dead code elimination [14] to avoid having unnecessary code in the organ when transplanted in the host.

The above mentioned ideas would make the multilingual organ as close as possible to the human written organ and would improve maintainability and readability of the transplanted organ. Another maintainability cost is about being able to keep the multilingual organ in sync with new updates of the donor system. Section 2.8 reviews some ideas for this. The most of the research reviewed in Section 2.8 considers the case of the same language transplantation. Thus, those approaches would need to be updated to take into account the difference in languages.

A simple idea to tackle the problem of keeping the multilingual up to date would be to always redo the transplant when the donor system has major updates. Since all the manual inputs for the transplants where provided with the initial transplant, this would be a completely automatic process. Even if the time for a multilingual transplant is considerable, (as we so in Section 5.5.4) we believe is

[14]https://glean.software/

reasonable to rerun the transplantation process at major updates to the donor since this is a completely automated process and major updates do not happen very often.

5.8 Conclusion

We introduced multilingual software transplantation, the transplantation of functionality between programs written in different languages. We have presented τSCALPEL, a multilingual software transplantation tool. We have used τSCALPEL to show the feasibility of multilingual software transplantation: τSCALPEL successfully transplants and translates functionality from Java to Swift and from Fortran to Python over 10 different host–donor pairs.

Chapter 6

Future Work

This chapter suggests some future work areas for the research reported in this thesis. This thesis reports our results in automated program repair and the initial steps into a particular kind of automated program repair: automated software transplantation. There are a couple of areas of future work, out of which the automated software transplantation approaches might benefit, such as organ validation, maintainability and readability of the organs, or removing the burden of human inputs that the current transplantation approaches require. Our automated program repair work might investigate in future areas such as more complex fixes, attempt to fix different kind of errors, or investigating the impact in production of such automated fixes.

Our automated program repair work is the first time that automated end-to-end repair has been deployed into production on industrial software systems used by billions of people, but still much remains to be done in this space. In this work we tackle only simple fixes to null pointer exceptions (NPEs). Our fixes ameliorate the problem by making the crash don't manifest. Future work might explore fixing the actual root cause of the crash. Such more complex fixes must consider their feasibility in practice in terms of time to identify a good fix and in terms of resource usage at the scale of industrial systems consisting in millions on lines of codes.

Future work in automated program repair might also attempt fixes to other kind of errors. In our case we identified about half of all the crashes as being NPEs, but there are also other kinds of crashes that might be amendable to automated program repair, such as the case of class not found crashes, caused by a change making the compiler not including a class in the final binary. More than crashes, future work might attempt to fix other kind of problems, such as performance regression.

In our automated program repair work we investigated the state of a fix patch until being landed in production. This involved a real developer looking at the fix and accepting it. But what about the impact of the fix in production? Has the fix really fixed the crash without introducing other new problems? Future research might consider this aspect.

Finally, our approach for automated program repair involved 3 fix strategies: template fix, mutation fix, and diff revert. It would be interesting in future research to try in parallel multiple state of the art automated program repair approaches (such as GenProg [185]) to compare them in terms of the capabilities to produce a patch that a human code reviewer would accept, time required to produce the patch, and the resource usage.

As in the case of automated program repair, much remain to be done in the future in the area of automated software transplantation. A first area of future work would be extending the validation of our monolingual software transplantation approach to extensively validate the capabilities of μSCALPEL at doing software transplantation between two different, unrelated software systems. The extension could include a set of random experiments: one could randomly selecting open source project from open source repositories such as GitHub, SourceForge, or Bitbucket, select one to be a host, the other to be a donor, and randomly select a feature for transplant. This will show the capabilities of μSCALPEL to be applied in real world situations, and in different context, on real world programs. An interesting case study for this extension would be autotransplantation for autotransplantation: use μSCALPEL for automated transplantation of the genetic programming algorithm into the ROSE version of μSCALPEL (Section 4.5).

The organ validation step and the organ adaption stage, for both monolingual and multilingual software transplantation are currently relying on test suites that the human developers provide to μSCALPEL and to τSCALPEL. We believe that these test suites can be automatically generated and extended with verification approaches. An idea to automatise the test suite generation is the co-evolution [18] of the organ's test suite, together with the organ itself. Here we will not only just evolve organs to pass the organ's test suite, but we will automatically evolve the organ's test suite, for capturing bugs in the organs, or for achieving high code coverage (*e.g.* branch coverage). Because of this, we will have more evidences for the correctness of our organs, since them passed an improved organ test suite. Here again, the evolved test suite might be used in different contexts, such as manual transplantation. The model we will follow here is predator—prey: the organ is the prey, while the organ's test suite is the predator, aimed at '*killing*' an organ, by catching bugs. Under these assumptions, even if the users of our approach do not trust the automatically done transplants, the test suite generated might be still useful for a human version of our transplants, or for the host system. Another idea we might explore is to adapt the test cases that we have for the host program and that exercise the organ, to the organ in the donor program.

To extend our testing-based approach with verification techniques, we can think at the organ in the terms of execution traces. We can view the traces as strings, and then the unvisited traces (from the existing test cases) share prefixes, suffixes and subsequences (for loops) with the visited one. The question to answer here is what we can formally prove about the unvisited traces, and what we can generalise from the set of traces that a test suite generates to the other traces, not exercised by the test suite. For answering to these questions, we might assume knowledge about the oracles and input domains, such as: distribution of the input domain, or test suite adequacy.

Another verification approach is probabilistic verification [171]. Here we can assume that the organ unit test has a uniform distribution over the organ's input domain. The organ might be reticent, and might have a behaviour whose likelihood is rare. We assume that we can parametrize the likelihood of the rare behaviour. Here the question is: what degree of reticence can μSCALPEL and τSCALPEL tolerate? For exploring this, we plan to build artificial organs, hosts, and donors, aimed at stressing our autotransplantation tools. We will make hypothesis about the degree of tolerance in hidden behaviour and experiment with different categories of host, donors, or test suites, for accepting or rejecting our hypothesis. For example we might compare what happens when the organ test suite exercise the hidden behaviour, compared to what happens when the organ test suite does not exercise the hidden behaviour. We can than check how many times the transplant was correct, even if the hidden behaviour was not tested by the organ test suite.

As an example, consider the following code: `if (x==10){return true} else{return false}`. In this case we can parametrize the input domain for the variable '`x`', and explore: the impact of the spike on the fitness components; the impact of the test suite's knowledge about the spike; the impact of the probability of the spike (organ's reticence).

Any of the above mentioned ideas we will decide to follow, it is very important for our research to get feedback about the organs, from human users of the beneficiary program. A few ideas about this are: design experiments where we transplant some small things, and ask other people to evaluate the resulted organs; release Kate plugins without specifying that them were automatically generated (which was already done, we just accept the response from Kate's development forum); make useful transplants into tools that we use daily, such that we can evaluate the transplants, without even thinking about this; report a manual analysis of the transplants, where this is possible (if the transplants are not very big).

Currently we have explored only the transplants for which the organ, once correctly identified, extracted from the donor, and translated into host's environment, was capable to pass al the test cases in the donor's test suite. However, this is not always possible in general. We call these cases '*Near – Miss – Transplants*', and will explore them in a future paper. Near–Miss–Transplants are the transplants where the most of the test cases pass, but not all of them. Here we might explore mutation operators used in mutation testing [148] literature, or try to combine code from multiple donors, for passing the failed test cases. Another idea here is to evolve the test suite, such that the organ to be able to pass all the test cases in the evolved test suite. For this, the user of μSCALPEL might specify a priority on the test cases, and some test cases that are desirable to pass, but not necessary required.

In another contribution we might explore the evolution of an organ, from multi donor programs. We might start with an empty host program, and evolve complex functionality, from multiple donor systems. In future studies we might explore the vein's choosing impact, automatically identify host target implantation point, or use the work in feature location for automatically identifying possible donor annotations. Under these problem solved, the transplantation will be done completely automatic: μSCALPEL and τSCALPEL will not require any more the current required annotations: the implantation point in the host, and the entry point of the organ in the donor system.

In future work we might also explore topics related to what happens to the organ after the transplants. A future paper might tackle the problem of maintaining the organ in-sync with its implementation in the host, where we could explore techniques discussed in Section 2.8. Future research in automated software transplantation could explore the maintainability and the readability of the organs and try to improve them.

Another potential future work area for our work in software transplantation is to use our autotransplantation approach to extract the organ as a library, rather than transplanting the organ in a host. The advantage here would be that the organ would be generic and easy to instantiate in different programs that require its functionality. The extension to our autotransplantation tools (μSCALPEL and τSCALPEL) would be trivial to support this approach: rather than inferring the organ's type signature from the implantation point in the host we could allow the user to specify the desired type signature of the organ as a library.

Chapter 7

Conclusion

This chapter concludes this PhD thesis that reports our automated program repair approach implemented in SapFix; and our monolingual and multilingual automated software transplantation approaches that are a particular kind of automated program repair that repairs the functional characteristics of the host system by augmenting it with a missing but required functionality.

For the automated software transplantation work we have introduced: μTrans, our algorithm for monolingual software transplantation, that we implemented in μScalpel; and τTrans, our algorithm for multilingual software transplantation, that we implemented in τScalpel. Our approaches to autotransplantation combine static, dynamic analysis, and genetic programming to extract, modify, and transplant alien code from a donor system into a host system both between the same programming language, or across different programming languages. The SapFix work reports the first automated program repair technique used in industry, at scale, for real world systems of millions of lines of code that are used by over two billion people.

Our exploration of the relevant literature clearly showed the need for both these contributions: the GP/GI literature for automatically optimising software systems treated in general only non-functional properties of a software system (with very few exceptions [286, 288]). The current automated program repair literature does not provide any example of an automated program repair system that scales at millions of lines of codes in products used by billions of people such is the case of SapFix.

We aim at providing all the tools that we develop as open source projects. Currently we open sourced our monolingual software transplantation tools and experiments (`http://crest.cs.ucl.ac.uk/autotransplantation`). Since in some cases the installation process is difficult (especially because of the use of the ROSE compiler infrastructure), we want to make the job of the users of our tools very easy. Because of this, we dockerized all our monolingual transplantation tools and experiments, and provided access to Docker containers with them. Like this, if anyone wants to explore our projects, they must just run one script, and the entire environment is automatically set up.

The results reported in this thesis for the autotransplantation reports are very promising (Section 4.7, Section 4.9, Section 4.11 for monolingual; Section 5.5, Section 5.6). We validated our results, by using regression, augmented regression, and acceptance test suites. We analysed in detail 3 monolingual software transplantation in 3 case studies (Section 4.9, Section 4.11); and all our multilingual software transplantations in 10 case studies (Section 5.6).

Autotransplantation is a new (and challenging) problem for software engineering research, so we did not expect that all transplantation attempts would succeed. For monolingual software transplantation, we reported an empirical study, where we systematically autotransplanted five donors into three hosts. 12 out of these 15 experiments contain at least one successful run. μSCALPEL successfully autotransplanted 4 out of 5 features, and all 5 passed acceptance tests. In all, 65% (188/300) runs pass all tests in all test suites, giving 63% unanimous pass rate; considering only acceptance tests, the success rate jumps to 85% (256/300). μSCALPEL and all our monolingual software transplantation experiments are available open source at `http://crest.cs.ucl.ac.uk/autotransplantation`. For the multilingual software transplantation approach, we reported an empirical study that consists in 5 transplants from Java donor into Swift hosts and 5 transplants from Fortran donors into Python hosts. Across these 10 different transplants, 72% (145/200) pass all our tests and are deemed to be successful under a manual analysis. For all of our Java to Swift transplants we report case studies, where we compared their outputs with the outputs of equivalent human written organs in the language of the host (*i.e.* Swift). For the case studies we uniformly randomly generated thousands of inputs, where the input space of the organs allowed, under which, the behaviors of the transplanted organs were equivalent with the behaviours of the human written organs in all the cases. Where random inputs were not possible, we manually generated inputs to asses how the multilingual transplants compare with human written organs.

To show that our autotransplantation approach might do useful transplants, we reported 2 different case studies. In the first one, we compared the autotransplantation of H.264 encoder feature, from x264 int VLC open source media player. This is a task humans had previously accomplished, by manual, and laborious work, extended in a time period of over 12 years. This study indicates that automated transplantation can be useful, since we showed μSCALPEL could do this task in only 26 hours, passing all regression and acceptance tests.

The second case study reports the automated transplantation of call graph drawing ability, and layout feature, both for C programs, from GNU `cflow` , and GNU `Indent` donors, into `Kate` text editor. The rich set of features and plugins make `Kate` a lightweight IDE for C developers, who identified these 2 functionalities (that we transplanted) as missing, and requested them on the `Kate`'s development forum. μSCALPEL required on average 101 minutes for transplanting the call graph generation feature, while transplanting the C layout feature took on average 31 minutes. The success rate for `cflow` donor is 80% (16/20), while the success rate for `Indent` donor is 90% (18/20).

The automated program repair tool SAPFIX has been deployed and is now running in the continuous integration system of Facebook. SAPFIX automatically produced and landed fixes into systems of tens of millions of lines of code that are used by hundreds of millions of people around the globe every day for communication, networking or community building. The entire process of program repair was automated: Sapienz automatically detected crashes and reported them to SAPFIX; and SAPFIX automatically produced candidate fixes, tested them, and finally landed them in production. This is the first time that automated end-to-end program repair has been deployed into production on industrial software systems.

This PhD thesis develops tools, techniques, algorithms, and theory, for the difficult problem of automated software transplantation and automated program repair. Even if our initial results are very promising, there is still a lot of work to do in these areas, as the future work ideas in Chapter 6 shows.

Appendix A

Grow and Serve: Growing Django Citation Services Using SBSE

Appendix B

Indexing Operators to Extend the Reach of Symbolic Execution

Indexing Operators to Extend the Reach of Symbolic Execution

EARL T. BARR, CREST, University College London, UK
DAVID CLARK, CREST, University College London, UK
MARK HARMAN, CREST, University College London, UK
ALEXANDRU MARGINEAN, CREST, University College London, UK

Traditional program analysis analyses a program language, that is, all programs that can be written in the language. There is a difference, however, between all possible programs that can be written and the corpus of actual programs written in a language. We seek to exploit this difference: for a given program, we apply a bespoke program transformation (INDEXIFY) to convert expressions that current SMT solvers do not, in general, handle, such as constraints on strings, into equisatisfiable expressions that they do handle. To this end, INDEXIFY replaces operators in hard-to-handle expressions with homomorphic versions that behave the same on a finite subset of the domain of the original operator, and return ⊥ denoting unknown outside of that subset. By focusing on what literals and expressions are *most useful for analysing a given program*, INDEXIFY constructs a small, finite theory that extends the power of a solver on the expressions a target program builds.

INDEXIFY's bespoke nature necessarily means that its evaluation must be experimental, resting on a demonstration of its effectiveness in practice. We have developed INDEXIFY, a tool for INDEXIFY and released it publicly. We demonstrate its utility and effectiveness by applying it to two real world benchmarks — string expressions in coreutils and floats in fdlibm53. INDEXIFY reduces time-to-completion on coreutils from Klee's 49.5m on average to 6.0m. It increases branch coverage on coreutils from 30.10% for Klee and 14.79% for Zesti to 66.83%. When indexifying floats in `fdlibm53`, INDEXIFY increases branch coverage from 34.45% to 71.56% over `Klee`. For a restricted class of inputs, INDEXIFY permits the symbolic execution of program paths unreachable with previous techniques: it covers more than twice as many branches in coreutils as Klee.

Additional Key Words and Phrases: Symbolic Execution, Reliability, Testing

1 INTRODUCTION

Symbolic execution (symex) supports reasoning about all states along a path. It is limited by its solver's ability to resolve the constraints that occur in a program's execution. Different types of symex handle intractable constraints differently. Broadly, static symbolic execution abandons a path upon encountering an intractable constraint; dynamic symbolic execution concretises the variables occurring in the constraint and continues execution, retaining the generality of symex only in the variables that remain symbolic. In this paper, we improve the responses of symex for a specific program, allowing it to continue past intractable constraints without resorting to fully concretising the variables involved in that constraint.

Our approach is INDEXIFY, a general program transformation framework that re-encodes intractable expressions, in any combination of types, into tractable expressions. This is impossible in general but can be achieved to a limited (and varying) degree for any particular program, so we characterise INDEXIFY as program-centric. To transform a program, INDEXIFY homomorphically maps a finite subset of the program's algebra of expressions to an algebra of indices (or labels), augmented with the undefined value ⊥.

Authors' addresses: Earl T. Barr, CREST, University College London, Malet Place, London, WC1E 6BT, UK, e.barr@ucl.ac.uk; David Clark, CREST, University College London, Gower Street, London, WC1E, UK, david.clark@ucl.ac.uk; Mark Harman, CREST, University College London, Gower Street, London, WC1E, UK, mark.harman@ucl.ac.uk; Alexandru Marginean, CREST, University College London, Gower Street, London, WC1E, UK, alexandru.marginean.13@ucl.ac.uk.

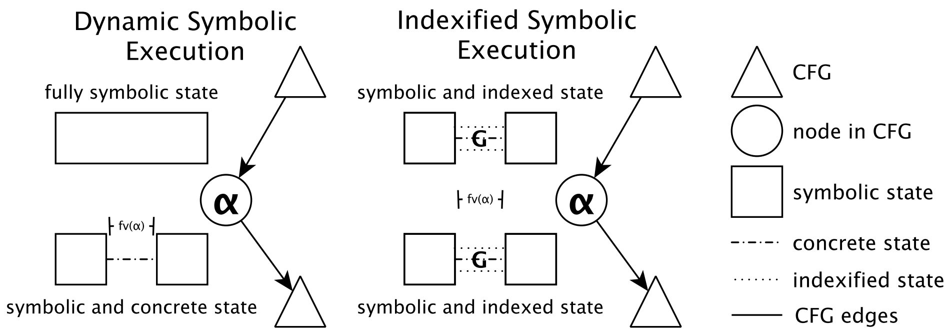

Dynamic Symbolic Execution
fully symbolic state
fv(α)
symbolic and concrete state
α
Indexified Symbolic Execution
symbolic and indexed state
G
fv(α)
G
symbolic and indexed state
α
CFG
node in CFG
symbolic state
concrete state
indexified state
CFG edges

Fig. 1. Standard dynamic symex compared to indexified symex; the control flow graph (CFG) node is a control point and the solver returns unknown on queries containing α.

Indexify is a program transformation that rewrites its input program to replace operators with versions that 1) are restricted to G, a finite subset of their original domain and range, and 2) take and return indices over G. Let *indexOf* map G to $\mathbb{N}$. When Indexify replaces the operator f, its finite replacement $\hat{f}(indexOf(x)) = indexOf(f(x))$ if $x \in G$ and $\perp$ otherwise. To build $\hat{f}$, Indexify memoises the computation of f over G. To this end, Indexify takes an input P, a (small) set of types $\mathcal{T}$, and (small) set of literals, S, of type $\mathcal{T}$, as seeds. In this work, we harvest constants from the program as seeds, as we explain in Section 5.1. It takes two sets of operators B and F, not necessarily distinct, where F's elements occur in P and may produce intractable constraints. Indexify transforms a program P in stages. First, Indexify repeatedly and recursively applies the operators in B to expand S to a larger set, G, the "Garden". Then, it memoises $f \in F$ over G to produce $\hat{f}$. Finally, it rewrites P to use the memoised versions of F.

Consider Figure 1, which depicts dynamic symbolic execution (DSE). In the figure, α is intractable, so the solver returns unknown when queried about α. On the left, DSE replaces the free variables in α with concrete values, either drawn from a concrete execution that reaches α (concolic [Marinescu and Cadar 2012]) or generated using heuristics [Chen et al. 2013], collapsing the state space of α's variables to those concrete values, but allowing symex to proceed over the rest of the state space.

On the right, we see Indexify in action on a transformed program P. Over α's free variables, the program is restricted to the garden G. An indexed expression, like α, that contains $\hat{f}$ evaluates to $\perp$ when it takes as an argument either $\perp$ or an unindexed value or it evaluates to an unindexed value. Indexed operators like $\hat{f}$ are lookup tables that generate constraints in equality theory as we show in Section 4.3. Indexify's transformation permits symbolic reasoning over the algebra of indices at the cost of explicit ignorance, reified in $\perp$, about values outside the index set; it allows the symbolic exeuction of the the indexed program over a (previously intractable) subset of the original program's state space, permitting the symbolic exploration of previously unreachable regions of that state space. If we indexify the problematic operators in α, symex can continue, restricted to G, past α in Figure 1. Once we apply Indexify to a program, the problem becomes identifying a useful set of expressions whose indexification might improve our ability to find bugs.

An indexed program under-approximates the semantics of the original program in the following way: if a transformed program's execution stays entirely within the indexed subset of values and operators, its output is either an index that is an image under the homomorphism of the original program's output or it is undefined (returns $\perp$). The under-approximation of the semantics and its dependency on the choice of the algebra to indexify again emphasises the

```
47  + typedef enum {NVidia, NVidiaCorporation, ...} string_enum;
48  - static const char * Nv11Vendor = "NVidia␣Corporation";
49  + static const string_enum Nv11Vendor = vendor11;
50
51  BOOLEAN IsVesaBiosOk(...){
52     ...
53  -  if (!(strncmp(Vendor, Nv11Vendor, sizeof(Nv11Vendor))))
54  +  if (!(i_strncmp(Vendor, Nv11Vendor, sizeof(Nv11Vendor))))
55  -    assert(strncmp(Vendor, Nv11Vendor, MAXLEN) == 0);
56  +    assert( i_strncmp{(Vendor, Nv11Vendor, MAXLEN) == 0);
57  }
```

Fig. 2. Diff of buggy code from ReactOS, after Indexify: the assertion, which we added to reify the bug, at line 55/56 can fail; this bug is difficult to reach under either pure or concolic symbolic execution; Indexify reaches and triggers the bug by restricting variables to a finite set of indices (adding line 47) and by replacing types, constants (line 48/49), and operators (lines 53/54, and 55/56) to work with these indices.

```
Garden:     G = {∅, N, V, i, d, ..., NV, aN, ...}
IOT:        int i_strncmp(const string_enum lhs, const string_enum rhs, int count){
                if(lhs == vendor && rhs == vendor11 && count == 4 return 0;
                else if(lhs == vendor && rhs == vendor11 && count == 8) return 1;
                ... return -1; // -1 represents ⊥
            }
```

Fig. 3. The garden, and the indexed operator tables IOTs (indexed initial operators) that Indexify generates for the code snippet in Figure 2. We use the Kleene Closure to build this example: G is the Kleene Closure of all the strings in ReactOS. We build the IOT `i_strncmp` for the operator `strncmp`. Section 4 describes how Indexify achieves the above.

bespoke nature of the transformation and its dependence on the goodness of the choice. As we demonstrate later through examples, in practice, it is not difficult to make a good choice. Our main contributions follow

(1) The introduction and formalisation of a general framework for restricting operators to produce tractable constraints, allowing symbolic execution to explore some paths previously only concretely reachable (Section 3);
(2) The realisation of our framework in the tool Indexify (Section 4) available at <url>; and
(3) Comprehensive demonstrations of Indexify's utility (Section 6): we compare Indexify with dynamic symbolic execution, *i.e.* Klee [Cadar et al. 2008], and concolic testing, *i.e.* Zesti [Marinescu and Cadar 2012]. We show that Indexify achieves 66.83% branch coverage, compared to Klee's 30.10% and Zesti's 14.79%, on coreutils, and does this within less than a third of the time that Klee without Indexify requires on average. Finally, we show it reaches bugs that Klee alone does not on a famous C bug finding benchmark.

2 MOTIVATING EXAMPLE

Figure 2 presents a code fragment that contains a real world bug in `ReactOS` [Developers 2016b], an open-source operating system. Commit `5926581`, on 21.12.2010, changes the type of variable `Nv11Vendor` from array to pointer. This causes the `sizeof` operator to return the size of a pointer, not the length of the array. ReactOS developers fixed this bug in commit `30818df`, on 18.03.2012, after ReactOS's developer mailing list[1] discussed it. The developers fixed the bug one year and three months after it was committed.

[1]https://reactos.org/archives/public/ros-dev/2012-March/015516.html

Figure 2 shows only the relevant code, after using Indexify, in diff format. Although Indexify operates at `LLVM` bitcode level, we use source code here for clarity. Indexify adds line 47 to define indices as an `enum`. Figure 3 shows the garden, and the indexed version of the operator `strncmp` (`i_strncmp`). More details about the garden, the indexed operators, and how we build them, are in Section 4.

The bug is a classic error: line 57 (lines 53/54 in Figure 2) in `vbe.c` of ReactOS, commit `30818df`, incorrectly applies `sizeof()` to a string pointer, not `strlen()` [Wagner et al. 2000]. As a result, `strncmp()` compares only the first 4 characters of its operands, assuming a 32 bit pointer. A pair of strings, whose first four characters are identical then differ afterwards in at least one character, triggers the bug when bound to '`Vendor`' and '`Nv11Vendor`': the `if` condition on lines 53/54 wrongly evaluates to true and assertion on lines 55/56 fails. This bug is an error in a string expression and the theory of strings is undecideable, when the string length is unbounded [Quine 1946]. Thus, most string solvers return UNKNOWN on this constraint [Bjørner et al. 2009].

Static symbolic execution must content with intractable constraints. CBMC, for instance, errors on them [Kroening and Tautschnig 2014] and would not reach the assertion on line 55. Klee [Cadar et al. 2008] implements dynamic symbolic execution and uses bit-blasting. When Klee reaches the `if` on line 53, Klee internalizes `strncmp` to bitblast it. The `strncmp` function loops over the length of strings, causing Klee's default solver to time out with its default settings (1 minute time-out). Thus, Klee does not produce an input that triggers this bug. Concolic testing searches a neighbourhood around the path executed under its concrete inputs. Upon reaching exit, concolic testing backtracks to the nearest condition, complements it, then restarts execution from the entry point [Godefroid et al. 2005; Sen et al. 2005a]. Thus, concolic testing can reach this bug only if it is given a concrete input in the neighbourhood of this bug. Unlike concolic testing, which is tethered to a single concrete execution, Indexify can symbolically reason over all the values in its finite set G, broadening its exploration relative to concolic testing.

Indexify finitizes operators in undecideable theories by transforming them into finite lookup tables over values of interest thereby converting potentially undecideable expressions into decideable ones. To build G, the set of interesting values, Indexify harvests the constants in a program, such as the ones in Figure 2, as seeds, then concretely and repeatedly applies operators, such as Kleene closure [Kleene 1951], to the constants, up to a bounded number, to expand the set. From the constants in Figure 2, this process produces `NVidia` and `NVidiaCorporation` among others. The lookup table for `strncmp` memoize the concrete results of repeatedly evaluating it on pairs drown from G. In Figure 2, to index the strings Indexify introduces the `enum` on line 47, then, on lines 48–49, it changes `Nv11Vendor`'s type to int and replaces the constant to which it is initialized to one of the indices introduced on line 47. Indexify then replaces `strncmp` with `i_strncmp` on lines 53/54 and 55/56. Indexing these and building `i_strncomp`, the memoised looking table for `strncmp` over the values in G is sufficient to violate the assertion at line 55/56.

Figure 3 shows the garden G: the extended set of constants that we consider in our analysis; and the indexed operator table (*IOT*) `i_strncmp`, the memoization table for the operator `strncmp`. We discuss these concepts in details in Section 3 and in Section 4. `i_strncmp` contains entries for all the values in G. If `i_strncmp` gets parameters that are out of G we abandon the path and return $\bot$. This means that the values that flowed into `i_strncmp` are out of our analysis.

Klee times out on the `strncmp` operator. When Klee runs on the indexified version of the code in Figure 2 it successfully executes the indexed `strncmp` operator: `i_strncmp`; and produces a bug triggering input. The bug-triggering constraint that Indexify generates is: Vendor $= 0 \wedge$ Nv11Vendor $= 1$. The solver produces the values: Vendor $= 0$ and Nv11Vendor $= 1$. Under these inputs, `i_strncmp` returns 0 signalling that the strings are equal. The assertion on line 56 fails, as the strings are equal only in the first 6 character but differ afterwards.

3 APPROACH

The concept of Indexify is quite general. It aims to transform a program so as to generate tractable constraints at *some* points at which it had previously generated intractable or undecidable constraints by restricting these constraints to a simpler theory over a finite set of values, augmented with the unknown value, $\perp$. The resulting, transformed constraints should be satisfiable whenever the original constraints were satisfiable, be more tractable with regard to satisfaction, and sometimes be satisfiable when the original constraints were not; but only when restricted to the finite set of chosen values and operators; crucially, SMT solvers can handle them efficiently. This simpler theory turns out in practice to depend on the way in which Indexify is implemented, although in each case the overall approach is the same.

To reiterate: for a given type or set of types we identify a useful set of literals and a desired set of operators on the literals, then memoize the outcomes of all combinations of applications of the operators on the literals — but only up to a limit, k, on the number of applications in any one expression. This can be represented simply as a finite set of index tables, one for each operator of interest. Since the useful set of literals is not necessarily closed under applications of the desired set of operators, we need to enter *undefined* for some entries in the tables. The final step is to perform a program transformation by replacing all the members of the memoized sets of literals and operators, as they occur in the program syntax, with their indexified versions. The effect from the point of view of constraint solving is that of shifting between logical theories. Solving constraints containing indexified operators can, as a result, use a more tractable theory such as equality.

This approach can be applied to any source theory, but, for the purposes of presentation and examples in the present paper, we restrict ourselves to `string` expressions and their operators. All non-trivial fragments of theory of strings are NP-complete [Jha et al. 2009], and thus, string constraints are intractable, making Indexify highly useful.

3.1 Terminology

Logical theories can be viewed as algebras. In universal algebra, an algebra is an algebraic structure, that is, a set of literals and a set of operators on the literals, together with a set of axioms that collectively play the role of laws for the algebra. Sometimes the notion of an algebraic structure is simplified to just a set of operators and the literals appear as nullary operators. In what follows, we explain in detail the soundness requirement for the transformation in the program syntax, i.e. that it must be a partial homomorphism between two algebraic structures. This does not map logical laws between the algebras. In our setting, there is not necessarily a homomorphism for the logical laws; to see why consider that laws of the naturals and strings.

To elaborate, in order to be sound, we require that the result of applying a transformed operator to transformed arguments yields the same result as applying the original operator to the original arguments and then transforming the result, whenever the result is defined in the transformed program. This is the minimum guarantee we should expect. Without it we could (unsuccessfully) transform any type to any type, any operator to any operator, e.g. strings to integers and replace operations on strings with operations on integers arbitrarily. In this section, we specify the behaviour of the homomorphism on the operators, then show that the algorithm for the transformation constructs a homomorphism of this kind. Finally, the transformation is implemented as a rewrite system on the syntax of the program.

It would be useful to be able to show that, whenever a transformed constraint has a model in the target theory, the untransformed version either has a model (but not necessarily the same model) in the source theory or is not satisfiable in the source theory. A proof of this would rely on properties of the individual theories and is left outside of the scope of this paper. Intuitively, given the homomorphism and given that the target theory is generally much simpler than the source theory, we believe that this will in general hold.

Considering the set of literal values and the set of operators on them that may occur in a program, we can partition each set into those that we indexify and those that we do not. Those that we do not indexify are left untouched by the transformation and on these the homomorphism is simply the identity. For simplicity, we require that there is no interaction between the untouched parts and the transformed values and operators. In consequence, once we identify a set of operators to indexify, we must also indexify a "sufficiently large set of values" which are of the input and output types for this chosen set of operators. We could make other choices with regard to the relationship between the indexified and unindexified operators and values but this is the simplest choice.

Constructing a "Garden" of Literal Values to Indexify: Here, we formallly present how we target a set of type literals and a set of operators to expand the initial set (seeds) into a larger set of literals G (garden).

We begin our description with some useful notation for types and operators. Let X be a set of operators. Denote the subset of nullary operators (literals) of X as X_0 and the set of non-nullary operators by $X_+ = X - X_0$. We will henceforth use the subscripts 0 and + to indicate sets of literals and sets of non-nullary operators respectively. Let H be a function that takes a set and returns a set of the same type. Use $H^k(X)$ to mean the recursive application of H k times to the set X, so that $H^0(X) = X$ and $H^k(X) = H^{k-1}(H(X))$.

Suppose we have a program P and want to indexify some of the operators that occur in P. Let T be the set of types in P and $\oplus$ be the set of operators used in P. Initially we have in mind a set of (non-nullary) operators of interest, ones out of which are presumably potentially problematic for SMT solvers. We first select a set, $S = B_0$, of nullary operators (literals) whose types are basic types that include all the argument and return types of these operators of interest. We call this set the seeds. Then we select B_+, a set of non-nullary operators on these seeds that we use to build a larger set of literals, the garden G. B_+ is not restricted to $\oplus$ and does not necessarily contain any of the operators of interest. Each literal in $S = B_0$ has type $\mathcal{T}$, where $\mathcal{T} = \tau_1 \uplus \tau_2 \uplus \cdots \uplus \tau_n$. In other words, $\mathcal{T}$ is a disjoint union of the types of nullary operators and each literal has one of those types.

Let $f \in B_+ \Rightarrow f : \mathcal{T} \to \ldots \to \mathcal{T} \to \mathcal{T}$, *i.e.* the argument and return types of f are in $\mathcal{T}$.

We define a function, $H : \mathcal{T} \to \mathcal{T}$ on a set of nullary operators, Z, as follows.

$$H(Z) = Z \cup \{f(x_1, x_2, \ldots, x_n) \mid f \in Z_+, x_i \in Z_0, \hat{f}(x_1, x_2, \ldots, x_n) \text{ is defined}\} \neq \bot$$

For simplicity of presentation, we have ignored all the "side information" about elements of Z as arguments to H in the type of H so as to focus on its application as an iterative step in growing the garden. We can then define G, the garden resulting from k applications of H, as

$$G = G_k = H^k(B_0) \tag{1}$$

Assuming $\mathcal{T} = \mathcal{T}'$ and that every possible non-nullary operator application to nullary operators is defined and returns a fresh literal, G grows quickly:

$$|G_k| = \sum_{m=2}^{k+2} \sum_{f \in B_+} \binom{|G_{m-1}|}{arity(f)} - \binom{|G_{m-2}|}{arity(f)} \tag{2}$$

Despite this prodigious growth rate, Indexify works well in practice given a small B, as we show in Section 6.1. One reason for this is that, in our experiments, only 2% of the applications of $f \in B_+$ produced a new value.

Indexifying a Set of Operators:

$$\begin{aligned}
&\tau \in \mathcal{T} && \Rightarrow \text{“}\underline{\tau\ x}\text{”} && \rightarrow \text{“}\underline{int\ \hat{x}}\text{''} && (3)\\
&l \in G && \Rightarrow \text{“}\underline{l}\text{”} && \rightarrow \text{“}\underline{\delta(l)}\text{”} && (4)\\
&l \notin G \wedge \text{ typeof}(l) \in \mathcal{T} && \Rightarrow \text{“}\underline{l}\text{”} && \rightarrow \text{“}\underline{\perp}\text{”} && (5)\\
&f \in F_+ && \Rightarrow \text{“}\underline{f}(a_1, \cdots, a_i, \cdots)\text{”} && \rightarrow \text{“}\underline{\hat{f}}(a_1, \cdots, a_i, \cdots)\text{”} && (6)\\
&\hat{f} \in \hat{F}_+ \wedge \exists a_i \text{ s.t. } \neg\delta^?(a_i) && \Rightarrow \text{“}\hat{f}(a_1, \cdots, \underline{a_i}, \cdots)\text{”} && \rightarrow \text{“}\hat{f}(a_1, \cdots, \underline{\delta_\perp(a_i),} \cdots)\text{”} && (7)\\
&f \notin F_+ \wedge \exists a_i \text{ s.t. } \delta^?(a_i) && \Rightarrow \text{“}f(a_1, \cdots, \underline{a_i}, \cdots)\text{”} && \rightarrow \text{“}f(a_1, \cdots, \underline{\delta_\perp^{-1}(a_i),} \cdots)\text{”} && (8)\\
&f \notin F_+ \wedge \delta^?(x) \wedge \nexists a_i \text{ s.t. } \delta^?(a_i) && \Rightarrow \text{“}\hat{x} := \underline{f(a_1, \cdots, a_n)}\text{”} && \rightarrow \text{“}\hat{x} := \underline{\delta_\perp(f(a_1, \cdots, a_n))}\text{”} && (9)\\
&\hat{f} \in \hat{F}_+ \wedge \neg\delta^?(x) \wedge \nexists a_i \text{ s.t. } \neg\delta^?(a_i) && \Rightarrow \text{“}x := \underline{\hat{f}(a_1, \cdots, a_n)}\text{”} && \rightarrow \text{“}x := \underline{\delta_\perp^{-1}(\hat{f}(a_1, \cdots, a_n))}\text{”} && (10)
\end{aligned}$$

Fig. 4. Φ_S, Indexify's rewriting schema: $x \in F$; $\hat{x} \in \hat{F}$; $\hat{f} \in \hat{F}_+$. We underline the redexes.

Having described the construction of the garden of literals, G, that becomes indexified inputs and outputs for the indexified operators, we return to the set of operators of interest that occur in P and that we wish to indexify. Let F_+ be this set and let $F = G \cup F_+$ so that $G = F_0$, the set of literals or nullary operators of interest. Note again that $F_+ \cap B_+$ may be empty. *The index function*: specify $\delta : F \longrightarrow \widehat{F}$ as an isomorphism that maps operators in F to fresh names for the indexed version of the operator that takes and returns indices over G[2]. Generally, we cannot index all the operators in a program, because some variables or operators can be defined externally.

Figure 4 shows the rewriting schema that Indexify implements to transform an input program. For $x \in F$, let

$$\delta_\perp(x) = \begin{cases} i & \text{if } \delta_0(x) = i \\ \perp & \text{otherwise} \end{cases}$$

In Figure 4, each a_i is an argument expression and the $\delta^?$ function checks if its argument has been directly indexed via Equation 3 or converted to an index because Equation 7 has wrapped it in a call to δ; when e is an expression and $\hat{x} \in \hat{F}$, its definition is:

$$\delta^?(t) \begin{cases} \mathbf{T} & \text{if } t = \text{“}\hat{x}\text{”} \vee t = \text{“}\delta_\perp(e)\text{”} \\ \mathbf{F} & \text{otherwise} \end{cases} \tag{11}$$

Equation 3 indexes variables and function declarations, in the latter case through repeated applications on a function's arguments and its return type name pair. ϕ_S only changes the parts in the program that appear in redexes in Figure 4. For the rest of them, the identity is implicit. Equation 4 replaces literals in G with their index under the homomorphism ϕ. Equation 5 handles the error case, where a constant has an indexified type but is not in G by replacing it with $\perp$. For each function call on a function to index, *i.e.* $\forall f \in F_+$, Equation 6 replaces the call's function identifier with the name of its indexed variant.

Equation 7 to Equation 10 rewrite function calls, which lack type annotations. For this reason, they do not overlap with Equation 3. They handle flows between indexed regions of the programs and unindexed ones. Equation 7 wraps unindexed arguments to an indexed call in δ to convert them to indices. Equation 8 it is the complement of Equation 7: it unindexes indexed variables that flow into unindexed functions. There are four different combinations of the two conditions in Equation 7 and Equation 8. The two combinations that we treat are flows across indexed to unindexed

[2]We use δ to denote our indexing function, because "δεικτησ" means "index" in Greek and starts with δ; it replaces the self-explanatory, but longer and less elegant name *indexOf* we used in the introduction.

boundaries. The other two combinations do not require transformation as no flows across indexed and unindexed boundaries occur in them. Equation 9 indexes returns from unindexed ($f \notin F_+$) functions into indexed variables ($\delta^?(x)$). The conditions $\nexists a_i$ s.t. $\delta^?(a_i)$ and $\nexists a_i$ s.t. $\neg\delta^?(a_i)$ in these two rules sequence their application, so the last two rules only trigger after exhausting the the first six rules, as $\delta^?(a_i)$ checks if a_i has been already indexed. Without the ordering, the rules overlap and we would not be able to prove the confluence of Indexify's TRS in Theorem 1. Similar to the previous two rules, Equation 10 complements Equation 9: it unindexes returns from indexed functions ($\hat{f} \in F_+$) that flow into an unindexed variable ($\neg\delta^?(x)$). As in the previous case, we only need rules for two out of four condition combinations (of the first two conjuncts of the guard), as only in these two we have flows across indexed and unindexed regions.

The schema can interact. Consider Equation 8 and Equation 9. Assume we have a call to $f \notin F_+$ that returns into a indexed variable and takes two indexed arguments. Two applications of Equation 8 and one of Equation 9, in any order, would rewrite this call. Correctness requires Indexify's rewriting to be confluent.

Theorem 1 (Confluence). *All instantiations of the term rewriting schema Φ_S into term rewriting systems are confluent.*

Proof Sketch 1. *Of the rules in Figure 4, Equation 3 and Equation 4 share G; Equation 6, Equation 8, and Equation 9 share F_+; and Equation 7 and Equation 10 share $\hat{F}_+$. The other rules cannot overlap because their guards restrict them to distinct sets. Equation 3 and Equation 4 do not overlap because their guards partition G. Similarly Equation 6 is defined over a different part of F_+ than Equation 7 and Equation 8. The guards of Equation 7 and Equation 10 guarantee that they do not overlap: $\exists a_i$ s.t. $\neg\delta^?(a_i)$ for Equation 7 and $\nexists a_i$ s.t. $\neg\delta^?(a_i)$ for Equation 8. Finally, $\delta^?$ prevents Equation 8 from being applied to eligible calls until Equation 7 has rewritten every parameter within it. Thus, none of the rewriting schemas have overlapping terms on the left hand side and Φ_S is non-overlapping. Φ_S is also left-linear: no variable occurs more than once in the left hand sides of the rules in Figure 4. We use substitute to instantiate the term rewriting schema Φ_S into the term rewriting Φ_i for a particular program: $\forall r_S \in \Phi_S, \forall t \in \mathcal{T} : \Phi_i = \Phi_S \cup r_S[t/\tau]$. For example, let $\mathcal{T} = \{float, string\}$. Then we generate the following rewriting rules from the rewriting schema in Equation 3, like "float x" $\longrightarrow$ "int $\hat{x}$" $\wedge$ "string x" $\longrightarrow$ "int $\hat{x}$". Since Φ_S is left-linear and non-overlapping and the substitution does not violate either property, so is Φ_i. Rosen [Rosen 1973] proved that left-linear and non-overlapping systems are confluent and so, each Φ_i is confluent, so Φ_S is confluent.*

4 IMPLEMENTATION

Given a set of operators (or functions) that can form undecidable expressions, Indexify memoizes a finite part of their behaviour, maps the rest to unknown $\perp$, then transforms a program to use them. The resulting program produces decidable constraints using these indexed operators for a subset of its original state space. We implemented Indexify on top of LLVM [Developers 2016a] for the `C` language to produce LLVM IR for the symbolic execution (symex) engine Klee [Cadar et al. 2008] to execute. We use Klee's default solver, *STP* [Ganesh and Dill 2007], because Dong *et al.* showed Klee performs best with STP [Dong et al. 2015].

Figure 5 shows the architecture of Indexify. The two main components of Indexify are its **Indexer** and **Rewriter**. To use Indexify, the user specifies P, the program to indexify, and F_+, the signature of the functions (or operators) to index. It is the user's responsibility to ensure to specify F_+ so that it contains all the operators needed to guarantee that *indexify*(P) produces decidable constraints, at least along the paths the user wishes to explore. Directly specifying F_+ is tedious. Section 4.4 details how we enable the user to indirectly specify F_+ in a simpler way.

If the user does not provide S, the indexer populates S with constants in P. Then, it computes G, the set of values to index and builds indexed operator tables. The rewriter rewrites P's IR to use indexed operators and values, including

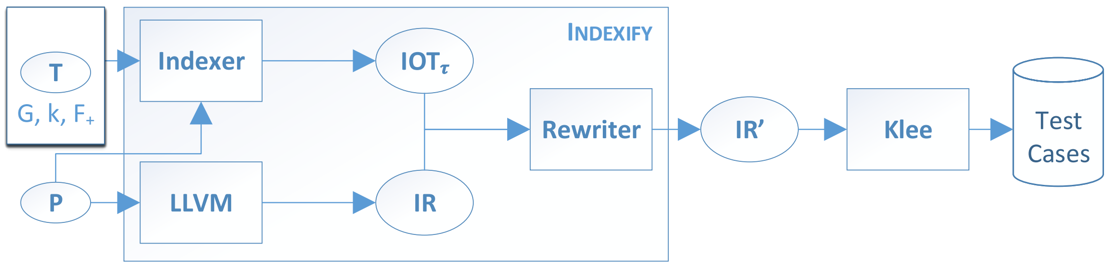


Fig. 5. The architecture of INDEXIFY; G, k, and F_+ are optional inputs.

Algorithm 1 *driver*$(\delta_0, B_+, F_+, k) = \delta'_0, \hat{F}_+$

This algorithm first calls Algorithm 2 to build the garden G, then calls Algorithm 3, using G (as carried within δ_0) to build the indexed operator tables.

Input: δ_0, an indexed set of nullary operators
B_+, a set of non-nullary functions for building G
F_+, a non-nullary set of functions to index
k limits applications of $b \in B_+$

Output: δ'_0, an extension of δ_0
$\hat{F}_+$, the image of F_+ under the homomorphism ϕ

```
1: $\forall i \in [1..k]$ do                          # Build the garden $G$
2:     $\forall b \in B_+$ do
3:         $\delta_0 := extend(\delta_0, b)$
4: $\forall f \in F_+$ do                             # Build $\hat{F}_+$
5:     $\hat{F}_+ := \hat{F}_+ \uplus memoise(\delta_0, f)$
6: return $\delta_0, \hat{F}_+$
```

constants, and to convert indices to values and vice versa. Finally, INDEXIFY runs Klee on the indexed IR to produce tests. Next, we discuss the operation of the indexer and the rewriter in detail, then explain how INDEXIFY produces constraints and close with its usage.

4.1 Indexer

For each $\tau \in \mathcal{T}$, we must index a subset of τ's values (G_τ), but which subset? INDEXIFY is *useless* without a good selection of values from τ. For instance, if all the values in G_τ take the same path (*e.g.* immediately exit), we learn nothing about the program under analysis. To build G_τ, the indexer uses functions from a set of constructors (B_+) to extend an initial set of seeds (S_τ); obviously, the quality of B_+ and $S = \cup_{\tau \in \mathcal{T}} S_\tau$ determine the quality of $G = \cup_{\tau \in \mathcal{T}} G_\tau$. In Section 6.1, we show that constants harvested from the source of the input program P make a surprisingly effective S and, in Section 5, that users unfamiliar with P do not improve on these constants much. Supplementing S with constants from developer artefacts other than the program text, like test cases or documentation, is future work. If the user does not specify B_+, the indexer defaults to using F_+, *i.e.* $B_+ = F_+$. Section 5.3 evaluates this choice of default for B_+. Adding new operators in B_+, to evaluate using INDEXIFY is straightforward: starting with the initial definitions, we automatically compute their indexed versions (Indexed Operator Table), by executing that operators, and memoising the results. We just require access to call the initial definition while memoising.

Algorithm 2 *extend*$(\delta_0, f) = \delta_0'$

Input: δ_0, an indexed set of nullary operators (constants)
$f : \tau_0 \to \tau_1$, a function to evaluate to extend δ_0
Output: δ_0', δ_0 extended using f

```
1: let $m := arity(f)$ in
2: $G := \{g \mid (g, n) \in \delta_0\}$                          # Snapshot $G$ before changing it
3: for all $g^m \in G^m \wedge typeof(g^m) = \tau_0$ do
4:     let $v := f(g^m)$ in
5:     if $\delta_\perp(v) = \perp$ then                         # Equation 11 defines $\delta_{\perp(v)}$
6:         $\delta_0 := \delta_0 \cup \{(v, |\delta_0| + 1)\}$
7: return $\delta_0$
```

Algorithm 3 *memoise*$(\delta_0, f) = \hat{f}$
This algorithm builds the image of f under δ_0.

Input: δ_0, an indexed set of nullary operators
$f : \tau_o \to \tau_1$, a non-nullary function to index
Output: $\hat{f}$, the image of f under the homomorphism ϕ

```
1: let $m := arity(f)$ in
2: $G = \{g \mid (g, n) \in \delta_0\}$
3: let $\hat{f} = \{\}$ in                                      # $\hat{f}$ is the image of $f$
4: for all $g^m \in G^m \wedge typeof(g^m) = \tau_0$ do
5:     let $v := f(g^m)$ in
6:     $\hat{f} := \hat{f} \cup \{((\mathbf{map}\ \delta_0\ g^m), \delta_0(v))\}$
7: return $\hat{f}$
```

Starting from S, Indexify first constructs G using the functions in B_+, as specified in Algorithm 1. It loops over number of applications k and over B_+, calling Algorithm 2. Later calls to Algorithm 2 are applied to the results of early calls. In this section, we split δ (Section 3) into δ_0 for the nullary operators and $\hat{F}_+$ for the rest of the operators. Algorithm 2 builds $\delta_0 = \phi|_{F_0}$ and Algorithm 3 builds $\hat{F}_+$.

Algorithm 2 simply enumerates G^m, filtering out type invalid permutations. In future work, we plan to experiment with sampling G^m. Given $S = \{a, b\}$, $K = 2$, and $B_+ = \{strcat, strstr\}$, Indexify produces $G = \{\emptyset, a, b, aa, bb, ab, ba\}$. Here, G contains all the values that can be obtained when applying the functions *strcat* and *strstr* twice first on S, then on S and the result of the first application. As discussed in Section 3, G grows quickly. Despite this forbidding growth rate, our experiments in Section 6.1 show that Indexify performs effectively when restricted to small values of k and $|B|$.

Next, Algorithm 3 indexes F_+, the functions operating over $\tau \in \mathcal{T}$. If the user inputs $\mathcal{T}$, we harvest F_+ from the intersection of functions used in the program, and a relevant library, *i.e.* `string.h`, when indexing strings, and `maths.h`, when indexing floats or double. For each $f : \vec{\tau} \to \tau \in F_+$, the indexer encodes $\hat{f}$ (or `i_f` in ASCII), the indexed version of f, as a sequence of if statements as follows:

```
i_f(τ x, τ y) {
   if x = 1 ∧ y = 1 return 1;
   if x = 1 ∧ y = 2 return 3;
   ...
   return -1; // return ⊥ when x ∉ G ∨ y ∉ G
}
```

Indexed operators (or functions) memoize the result of $f \in F_+$ on values in terms of the indices defined by δ. When the result of f's computation is not in G, $\hat{f}$ returns $\perp$. Thus, `i_f` above contains `if x = 1` $\wedge$ `y = 3 return 57;` because $f(\delta^{-1}(1), \delta^{-1}(3)) = \delta^{-1}(57)$. The number of if statements in an indexed operator is $|G|^n + 1$, but Sections 5.2 and 6.1 show experimentally that Indexify was effective using parameter settings that only doubled the size of its inputs on average. The indexer injects the definition of each $\hat{f} \in \hat{F}_I$ into P.

Algorithms 1–3 build δ and exactly realise Equation 1. Algorithm 1 is the driver algorithm that calls Algorithm 1 and Algorithm 2 to build $\delta = \delta_0 \cup \delta_+$ in two parts. Algorithm 2 simultanously extends δ_0 and constructs the garden G using the builders B_+ applied to G as it expands. Algorithm 1 calls Algorithm 2 only over B_+ on lines 1–3. Algorithm 2 calls each $b \in B_+$ on the values currently in G. If the resulting string is not currently in G, Algorithm 2 assigns it a fresh index and updates δ_0. Algorithm 3 constructs $\delta_+ = \hat{F} - \hat{F}_0$, the union of all the indexed operator tables for the functions in F_+. Algorithm 1 calls Algorithm 3 only over F_+ on lines 4–5. Algorithm 2 and Algorithm 3 perform no other computations.

4.2 Rewriter

The *rewriter* is a straightforward implementation of the term rewriting schema in Figure 4. We generate wrapper code that unindexes things on the fly. We initially index every occurrence of a variable of the indexed type, and further we unindex them on the fly, when the variables flow into unindexed operations. In the presence of aliasing, we unindex such values when they go into a different data type (structure), and later unindex it back, when it flows into an indexed data type / operator. To handle casts we use the index, and unindex function. When the program casts a indexed variable x to a different type, we first unindex x to the initial data type, and then we cast the variable of the initial data type, as in the initial program. Similarly, when the program casts a variable of an unindexed type to an indexed type, we index the result of the cast applied on the initial (unindexed) data type.

Figure 6 shows an example of applying ϕ_S. When G = { `"foobar"`, `"oobar"`, `"bar"`, `"oo"` }, Indexify replaces `"bar"` on line 6 (Figure 6a) with δ(`"bar"`) = 2 (Figure 6b), but ignores the constants `a` and `foo` on line 5 (Figure 6a). Equation 6 replaces `strstr` on line 6 in Figure 6a with `i_strstr` on line 14 in Figure 6b. Equation 8 unindexes indexed variables that flow into unindexed functions. In Figure 6b, Equation 8 unindexes the indexed variable `S1` on line 13, as the function `puts` is unindexed. Finally, Equation 9 indexes the return from unindexed functions into indexed variables in Figure 6b on line 11 since `strcat` is not indexed, but returns into `S2`, which is.

The rewriter uses `LLVM`'s API for IR manipulations and transformations[3]. In `LLVM`, types are immutable, so we cannot change them in place. Instead, Indexify outputs the indexed version of the IR of P in a new file. The *rewriter* is a visitor that walks the original IR of P: when it reaches an IR element of a type in $\mathcal{T}$, it creates a new instruction with $\mathcal{T}$ changed to `index`; otherwise, it simply echoes the instruction. In LLVM, a global variable and its initializor must have the same type and LLVM forbides casts or function calls in initializors. Thus, the visitor replaces values with indices in initializors. We execute Klee on this indexed version of the LLVM bitcode. To support indexing multiple types at time, Indexify keeps multiple gardens, one for each indexed data type. For example, one might want to index both the strings and the floats in a program. For this, Indexify keeps G_S, the set of string values in our domain restriction, and G_F, the set of float values in our domain restriction. The indexes in G_S and G_F do not overlap. Further, Indexify proceeds normally, but when indexing a type it uses indexes from the corresponding garden: G_S if the type to index is a string; G_F if the type to index is a float.

[3]http://llvm.org

```
1  int main(){
2      char S1[3], S2[5];
3      klee_make_symbolic(S1,
             sizeof(S1), "S1");
4      puts(strlen(S1));
5      S2 = strcat("a","foo");
6      if(strstr(S1, "bar")) return 1;
7      else return 0;
8  }
```

(a) P, the program to indexify.

```
1   int  i_strstr(int s1, int s2){
2       if(s1 == 0 && s2 == 2) return 2;
3       if (s1 == 0 && s2 == 3) return 1;
4       if (s1 == 3 && s2 == 2) return 0;
5       ... return -1; // return ⊥
6   }
7   int main() {
8       int S1,S2;
9       klee_make_symbolic(&S1, sizeof(S1), "S1");
10      puts(δ⁻1(S1));
11      S2 = δ(strcat("a","foo"));
12      if(i_strstr(S1, 2)) return 1;  // 2 = δ("bar");
13      else return 0;
14  }
```

(b) P' = indexify(P, string); `puts` and `strcat` are not in F_+.

Fig. 6. Indexification: Indexify also injects δ and δ^{-1} into P', although not shown.

Indexify allows an indexed operator to take both indexed and unindexed types. In this case, Indexify does not simply replace the function's body with an *IOT*; instead, it indexifies the function's body, replacing any internal calls to indexed operators with their indices versions and inserting index-casts to (un)index values, including the return value, as needed. Currently Indexify does not use Equation 5. Indexify does not need Equation 5 because for us S is the set of the literals in the program. Thus, all the literals in the program are in G.

4.3 Indexed Constraint Construction

To understand how an indexified program constructs constraints, consider the program P in Figure 6a[4]. Here, the string S is symbolic; and "bar" is a constant string. The call to `klee_make_symbolic` makes the program variable S symbolic. When G = { "foobar", "oobar", "bar", "oo" }, `indexify --type string P.c --garden path_to_G --F_+ path_to_F_+` produces P' in Figure 6b, whose behaviour is restricted to G over string operations. In P', $\hat{f}_{\text{strstr}} = i\_strstr$.

Indexify produces P'. When we symbolically execute P', we reach the call to `i_strstr` on line 12 (Figure 6b). For `i_strstr`, Klee delays calling the internal solver [Cadar et al. 2008], encoding `i_strstr` into an indexed operator table (*IOT*) as disjunctions for the branches of the `if` statement (Figure 6b). The constraints that `i_strstr` generates are: `(s1 = 0 ∧ s1 = 2 ) ∨ (s1 = 0 ∧ s2 = 3) ∨ (s1 = 3 ∧s2 = 2) ∨` *IOT*.

The constraints in the *IOT* of $\hat{f} \in \hat{F}_+$ are in the theory of equality by construction; an *IOT* is a lookup table that disjuncts equalities over the values of its parameters. These constraints are solvable by an integer solver equipped with equality theory in polynomial time. Barrett *et al.* [Barrett et al. 2009] show this and provide a linear time solving algorithm. Our implementation currently cannot fully exploit this fact because it extends Klee, which builds constraints directly in the bitvector theory with arrays. In our current implementation, Indexify first indexes the program then uses Klee for symbolic execution. Klee takes the indexed program and bitblasts it. In this case, Indexify still improves performance whenever bitblasting integers results in shorter constraints than bitblasting the initial data type. For example, a string of length N requires $N * sizeof$(char) bits. After indexing, the same string uses only *sizeof*(int) bits.

Indexify takes advantage of Klee's internal query optimisations. Klee removes unsatisfiable constraints from the path conditions via simplifications [Cadar et al. 2008; Dillig et al. 2010]. Under these optimizations, Klee selects only the relevant subset of $\hat{f}_{\text{strstr}}$ in the context of the predicate in the if statement on line 15 in Figure 6b: only the entries where

[4]This is an actual, if pedagogical example. We provide the constraints in SMTLIB format, as generated by Klee, at <url>.

the second parameter is $\delta(\text{bar}) = 2$. It does so prior to sending the query to the internal solver. Thus, the constraints that Klee actually sends to STP for `i_strstr` are: `(s1 = 0 ∧ s1 = 2 ) ∨ (s1 = 3 ∧ s2 = 2) ∨` $IOT|_{S_2=2}$.

Section 6.7 compares the number of clauses that Klee and Indexify generate on coreutils. Without optimisation, Indexify generates three times more clauses than Klee. The results drastically change, when we look at the number of clauses that reach the solver, after Klee's internal optimisations: indexified programs send $\frac{1}{10}$ as many constraints to the solver. Indexify's *IOT*, by construction, are particularly amenable to the syntactic constraint simplifications that Klee employs.

4.4 Usage

Once Klee is successfully installed, Indexify is easy to use: issuing `indexify --type string yourprogram.c` indexifies `yourprogram.c`, and then runs Klee on the indexified program. Indexify takes optional parameters. When runs with its `--outputIndexedIR` flag, Indexify outputs the indexed IR. By default, Indexify harvests S, the set of seed values to index from the input program. The switch `--addSeeds <file>` specifies a file containing seeds to add to the harvested constants; `--seeds <file>` specifies a file containing S and prevents constant harvesting. By default, Indexify automatically constructs the operator definitions, using Algorithm 3. The switch `--indexOpDefs <file>` specifies the `LLVM` bitcode file that contains indexed operators to allow their reuse across runs, amortizing the cost of their construction.

Specifying F_+, the functions in a program to index, is tedious. We allow the user to specify only $\mathcal{T}$, which triggers the indexer to populate F_+ from a header file. For $\mathcal{T}$ = `string`, Indexify indexes all the functions in `string.h`; for $\mathcal{T}$ = `float`, Indexify indexes all the functions in `maths.h`. For an arbitrary data type, the user needs to specify the name of the desired header file.

5 EXPERIMENTAL SETUP

Indexify operates on an input program P in two phases. First, its indexer constructs the "garden" of values G, then it builds operator tables for the target operators in P over G. Finally, Indexify transforms P to P' and symbolically executes it. As detailed in the previous section, the indexer takes four parameters: the types to index or a list of functions to index F_+, the seed values $B_0 = S$, the functions for building the garden from S, and k, a bound on the applications of the builder functions. In this section, we explore various setting for S, B_+, and k in order to fix them, leaving only F_+ to vary, in Section 6.

Corpus Our experimental corpus contains two benchmarks: `GNU Coreutils`[5], and `fdlibm53`[6]. GNU Coreutils contains the basic file, shell and text manipulation utilities of the GNU operating system, such as `echo`, `rm`, `cp` and `chmod`. We include all the 89 `Coreutils` in our experiments, and use Indexify (*string*) on them. We use the second benchmark, `fdlibm53`, for exploring the capabilities of Indexify, when indexifying floats. We picked these benchmarks because Indexify, Klee and Zesti can execute all the benchmarks in our corpus.

We select Klee (KLEE 1.1.0 47a97ce; LLVM 2.7), as a dynamic symbolic execution engine, and Zesti (the beta version on the Zesti's website[7]) as a concolic testing engine, for the comparison with Indexify. We selected these 2 tools, as Indexify is based on Klee, as is the case for Zesti. Indexify first indexifies the program, and further calls Klee on the indexified program. We also decided to compare Indexify with Klee and Zesti, because of the fact that we support

[5]http://www.gnu.org/software/coreutils/coreutils.html
[6]http://www.validlab.com/software/fdlibm53.tar.gz
[7]http://srg.doc.ic.ac.uk/zesti/zesti.tar.gz

symbolic execution on `C` code. Klee is the most famous symbolic execution engines for `C` code. We called all the three tools (Indexify, Klee, and Zesti) with exactly the same parameters as used in the initial Klee paper [Cadar et al. 2008].

5.1 Human-Provided Seeds

Indexify constructs G, the subspace of value to which Indexify restricts a program's behaviour, from S. Thus, defining S is crucial to Indexify's effectiveness. Indexify defaults S to the constants of $\mathcal{T}$ in P. This extraction method exploits the domain knowledge embedded in these constants to bias B, and therefore G, to values that P is more likely to compute.

Here, we ask whether human intuition can help Indexify by augmenting S with values that a developer considers interesting? To decide if human intervention in seed selection is effective, we compare the manual effort to discover seeds against the coverage and execution time gains of the augmented set of seeds. For this comparison, we uniformly selected 10 programs from `coreutils`: `cat`, `expand`, `fold`, `mknod`, `mktemp`, `runcon`, `shred`, `tsort`, `unexpand`, and `wc`. One of the authors spent no more than ten minutes to construct strings that he thought might allow symbolic execution of the indexed program to explore new paths or corner cases and added them to S. We ran Indexify with T set to strings, B_+ set to string functions occuring in P and uclibc, and $k = 3$ on this set.

The results were identical: the human-augmented seeds provided no discernable improvement. Given Indexify's overall effectiveness, we take this as a testament to the effectiveness of Indexify's constant harvesting heuristic.

5.2 Bounding k to Operator Chain Length

Indexify's core indexer algorithm (Algorithm 1) is computationally and memory expensive as a function of k, the number of applications of the functions in B (Equation 2). Setting k is a trade-off between the power of analysis (*i.e.* the size of G) and the computational cost of running Indexify. First, we identify the value of k in our corpus, then we observe the performance when using the identified k.

From our corpus, we infer k from the lengths of chains of applications over F_+; we define chains have nonzero length. In our first experiment, we statically symbolically execute our corpus to collect symbolic state, from which we extract operator chain lengths from def-use chains in killed expressions. Figure 7 shows the lengths of operator chains k in our corpus. The first boxplot, labelled *All.coreutils*, reports the distribution of all operator chain lengths in `coreutils`: the median is 11; the minimum is 1; and the maximum is 2833. The percentage of outliers is 11.66% upward and 0% downward, since 1 is the first quartile and we do not have 0s. A manual exploration of uniformly picked outliers reveals that loops cause them.

Figure 7 also reports boxplots for `string` and `float` operators. For `string` operators, the median value is 2, with 3.64% outliers; for `float` operators, the median value is 1, with 15.53% outliers. Setting k to the third quartile of operator chain lengths in our corpus reduces the probability that symbolically executing an indexified benchmark will escape G and bind $\perp$ to a variable, merely through operator applications. Thus, we fix $k = 3$.

5.3 Finding a Good Builder

Indexify uses the functions in B_+ to build the garden G from S. Which functions should we use? For strings, we consider two different builders. $B_+^* = \{^*\}$ contains only Kleene closure and $B_+^\circ = F_+$, the indexed functions in P. To build G, we apply the operators in B_+ up to k times. For example, let $S = \{a.b\}$ and $k = 2$. Under B_+^*, $G = \{\emptyset, a, b, aa, bb, ab, ba\}$. For floats, B_+° applies all the float operators in P including `maths.h` on all combinations of float constants in the program text up to k times. Each application of B_+ potentially generates a new value in G.

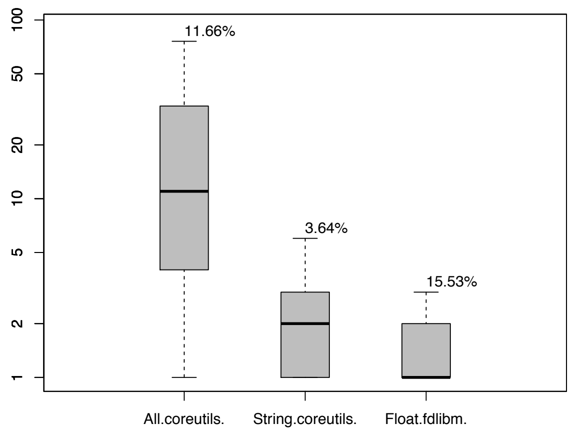


Fig. 7. Length of operator chains in coreutils; we use these values to set k.

Table 1. Measures of Algorithm 1 as a function of k.

	$k = 1$		$k = 2$		$k = 3$	
	B^*_+	B°_+	B^*_+	B°_+	B^*_+	B°_+
Time(sec)	324	123	544	263	1008	**363**
$\lvert G\rvert$	10792	69068	78920	163378	356760	263616
ICov(%)	26.22	52.21	34.09	55.65	55.32	**57.67**
BCov(%)	21.93	59.74	39.98	62.37	**69.73**	66.83

To compare B^*_+ and B°_+ on strings up to $k = 3$, we executed Indexify on coreutils, then ran Klee on the resulting indexed programs. We constrained the G that B^*_+ constructed: only adding a generated value to G if its length was less than or equal to 8. We experimented with different maximum sizes, up to 20. We observed that the branch and statement coverages are not improving any more when considering strings with lengths greater than 8. Our experimental corpus, `Coreutils`, does not computes during its execution strings longer than 8 character. This is because the most of the programs in `Coreutils` do not use very often the concatenation operator over strings `strcat`. This is the only string operator that increases the sizes of strings during execution. The most of the strings that the programs in `Coreutils` use are flags, or file names, which are in general short in size. For different experimental programs, higher values of maximal string lengths might be required. Table 1's "Time" row shows that B^*_+ executes more than three times slower than B°_+. At $k = 3$, B^*_+ covers 55.32% of the instructions in coreutils ("ICov"), while B°_+ covers 57.67% of them. The branch coverage ("BCov") is 69.73% for B^*_+ and 66.83% for B°_+. These results show a trade-off between execution time and coverage. B^*_+ covers more strings, increasing both *BCov* and the execution time. The difference in *BCov* is only 3% and B°_+ achieves 2% higher ICov. Thus, we set $k = 3$ and $B_+ = B^\circ_+$ in Section 6.

When using k greater than 3, we observed an increase in runtime and a decrease in code coverage. This is due to the fact that the number of conjunctions in the constraints that Indexify generates grows exponentially, and Klee's internal solver time-out is reached. When this happens, Klee abandons the path with the constraint that time-outs. We ran k up to 10. The branch coverage decreased smoothly from 69.73% when $k = 3$ to 65.00% when $k = 10$. The execution time increased smoothly from 1008 seconds when $k = 3$ to 3150 when $k = 10$.

Table 1 shows that, for both B_+° and B_+^*, coverage increases with k up to 3, but execution time does not become unreasonable. Even $k = 3$ generates an immense garden, as Equation 2, in Section 3.1 shows and $|G|$ in Table 1 confirms. How does INDEXIFY manage to be effective, despite Equation 2? We hypothesized that $k > 3$ is feasible because very few elements are unique in practice. Especially with strings, many operators return substrings of existing elements. We evaluated our hypothesis on B_+°. Our results show that from the total of generated values, only 1.97% are unique; only the unique values are indexed.

6 EVALUATION

Here, we demonstrate that INDEXIFY can improve existing automated testing techniques by trading space to reduce time and increase solution coverage. When used to generate test data, dynamic symbolic execution and concolic testing aim to achieve the highest structural coverage possible. Since the *theoretical* space complexity of INDEXIFY is worse than exponential, an essential goal of this evaluation is to demonstrate that its space consumption, in practice, is manageable. Furthermore, we evaluate the degree to which this manageable increase in space consumption reduces time and improves solution coverage. We evaluate whether INDEXIFY can catch bugs that are out of reach to traditional symex. Then, we assess INDEXIFY ability to make the constraints easier for the underlying SMT solver by restricting the domain of some operators. Finally, we evaluate INDEXIFY when indexing floats.

6.1 Trading Space for Time and Coverage

Table 2. Overall results for INDEXIFY (String). We fixed the bound $k = 3$; we used $B_+ = B_+^\circ$. The trends support our core insight: INDEXIFY increases memory consumption reducing execution time and increasing code coverage

Metric	INDEXIFY	Klee	Zesti
Time(min)	6.05	49.52	22.70
Memory(MB)	900.45	241.16	3000.00
ICov(%)	57.67	41.24	20.60
BCov(%)	66.83	30.10	14.79

INDEXIFY trades memory for reduced execution time and increased code coverage. Memory has become cheaper and more abundant, but our core algorithm consumes exponential memory in the worst case. Can we significantly reduce execution time or increase code coverage at reasonable memory cost?

To answer this question, we compare INDEXIFY to dynamic symbolic execution using Klee and to concolic testing using Zesti. Zesti uses concrete inputs to kick-off concolic testing. It searches a neighbourhood around the path executed under the concrete inputs. For each concrete input, Zesti follows the concrete execution that the input generates and records the path condition. When reaching exit, Zesti backtracks to the nearest condition, complements it, generates a new input that obeys the new path condition, and restarts execution from the entry point [Godefroid et al. 2005; Sen et al. 2005a].

We compare their performance in terms of run time, memory consumption, instruction, and branch coverages. We index strings in the input programs when running INDEXIFY, *i.e.* $\mathcal{T}$ = string.

To configure INDEXIFY, Klee, and Zesti, we use the same settings that Klee used on the coreutils benchmark [Cadar et al. 2008]. The time-out is 3600 seconds; the maximum memory usage allowed is 1000 MB; and the maximum time spent on one query is 30 seconds. Although the maximum memory consumption is 1000 MB, Zesti might use more memory than this, as it composes multiple symbolic execution runs. We use $k = 3$, as we explain in Section 5.2.

We construct *IOT* and G online. We automatically harvest the seeds S from each program's text. F_+ contains the functions involving strings in the input program; we use $B_+^{\circ,3}$ (Section 5) to build G.

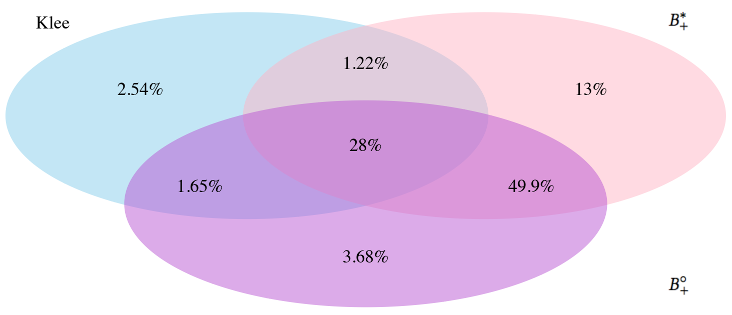


Fig. 8. Uniquely covered branches by: Klee, Indexify with B_+^*, and Indexify with B_+°

We report the instruction and branch coverage on `LLVM` bitcode, for the final linked bitcode file. This bitcode combines tool and library code. Achieving 100% coverage of this bitcode is usually impossible, because programs tend to use very little library code, relegating the rest to dead code. For example, `printf("Hello!")` does not cover `printf`'s format specification handling code [Cadar et al. 2008].

Table 2 reports our experimental results. As expected, these results (row "Memory") show that Indexify consumes more memory than Klee. Averaged over all coreutils, Klee requires 241.16 MB, while Indexify requires 900.45 MB. Indexify's memory consumption is reasonable relative to Zesti, consuming $\frac{1}{3}$ the memory (3000MB) Zesti does. Zesti exceeds the 1000MB memory limit on a single run of Klee, as it runs Klee multiple times.

Indexify has compensatory advantages to set against its memory consumption. First, Indexify improves time performance. Over all, Indexify requires 6.05 minutes mean time, while Zesti requires 22.70 minutes and Klee requires 49.5 minutes. Second, Indexify outperforms the existing state of the art in terms of coverage achieved: Indexify achieves 57.67% instruction coverage and 66.83% branch coverage. By contrast, Klee achieves 41.24% instruction coverage and 30.10% branch coverage while Zesti achieves 20.60% instruction coverage and 14.79% branch coverage.

While there is much debate in the testing community over the value coverage [Gligoric et al. 2013; Lakhotia et al. 2009], there is little doubt that greater coverage is to be strongly preferred over lower coverage. We argue that the increased coverage and reduced run-time are more significant than the increase in memory consumption. For an increase of 659.29 MB of memory for Indexify, we get 16% increase in instruction coverage; a 36% increase in branch coverage; and a 43 minute reduction in run time, on average.

6.2 Unique Statements Coverage

We explore how many previously unreachable branches Indexify brings in the reach of symex and compare how many branches Klee and Indexify uniquely cover. In Figure 8, we show the percentages of branches that B_+°, B_+^*, and Klee uniquely cover. Klee uniquely covered 3.02% of branches in `coreutils`'s IR. Both B_+° and B_+^* missed these branches: they are infeasible in the indexed programs, because our garden G does not contain the strings that these branches use. To cover these branches, a greater K might help.

Figure 8 shows that although 2.54% of statements are covered only by `Klee`, Indexify enables symbolic execution to cover far more previously out of reach branches: on total, 66.58% are only covered by B_+^*, or B_+°. B_+^* covers uniquely the most branches: 13%. B_+° covers uniquely 3.68%. More branches are uniquely covered by the B_+^* because the *Kleene*

Table 3. **Bugs that Klee, and Indexify can reach.**

Project	Bug	Size	Klee	Indexify
ncompress	stack smash	1935	NO	YES
gzip	buffer overflow	4653	NO	NO
man	buffer overflow	2805	NO	YES
polymorph	buffer overflow	240	YES	YES

closure contains strings that are not the result of operator applications. Some branches are uniquely covered by B_+° because K applications of string operators may lead to strings longer than the bound for the *Kleene* closure.

6.3 Bug Finding

The most compelling evaluation for any testing technique is the ability to reveal bugs that other techniques cannot. Accordingly, we investigate whether Indexify can identify bugs not detected by Klee.We evaluated Indexify on a set of bugs from *Bugbench*, a famous benchmark for C bug finding tools [Lu et al. 2005]. We consider only the bugs that the default oracle in Klee can detect — bugs that cause programs to crash. For Klee, experiments [Cadar et al. 2008] on `coreutils` set the time-out to 60 minutes. `coreutils` average 434 LOC. Our corpus averages 2408 LOC, a 5-fold linear increase. Although Klee does not scale linearly, we increased the time budget 5-fold to five hours, for both Klee and Indexify.

Table 3 reports our results: Indexify catches two bugs that Klee cannot catch: the bugs in `ncompress` and `man`; both Klee and Indexify catch the bug in `polymorph`. Neither tool catches the bug in `gzip`. Not catching a bug means that the tool reaches the five hours timeout without generating a test case to reveal the bug in our benchmark [Lu et al. 2005]. Our results show that Indexify reaches bugs previously unreachable for Klee in two cases out of four.

6.4 Domain Restriction

Indexify restricts the domain of operators involving $\mathcal{T}$ to a subset of the initial supported values, the one that G contains, to make them simpler: Indexify rewrites constraints involving types in $\mathcal{T}$ into the theory of equality over integers. We report the number of constraints on which `STP` times out: How many SMT queries return "unknown" before and after the application of Indexify?

When running Klee on all coreutils, STP time-outs on 354 constraints. After indexifying the corpus, this number drops to 0. In 53 out of the 89 coreutils programs, Klee times out before finishing; only 6 programs time out for Indexify. These results show Indexify generates indexed programs that produce simpler constraints than Klee does when run directly on the original program.

We then divided the 89 coreutils programs into 3 categories: projects with at least 5% of the expressions indexed after applying Indexify; projects with at least 10% of expressions indexed; and projects with at least 15% of expressions indexed. These 3 categories overlap, as we wanted to assess the performance of Indexify as a function of the percentage of indexed expressions. There were no important differences in the performance of Indexify between these three groups. Indexify is not dependent on the proportion of program expressions that it converts, but rather on how much of the code in the program the paths that become reachable due to indexing make available for symex.

6.5 Indexifying Floats

To assess how well Indexify performs when indexing data types other than strings, we indexed the type `float` in our second benchmark: fdlibm53. For B_+ we used the intersection between the functions involving `floats` in fdlibm53 and the ones in `math.h`. We fixed $k = 3$, as the third quartile value of k, the length of chains of floating point operators in our corpus (Figure 7).

Table 4. Overall results of our experiments with Indexify (float) and $K = 3$.

Metric	B_+°	Klee
Time(min)	10.45	0.84
Memory(MB)	71.98	6.85
ICov(%)	83.35	50.91
BCov(%)	71.56	34.45

Table 4 shows Indexify's general effectiveness. Indexify increased instruction coverage 33% (row "ICov") and branch coverage 37% (row "'BCov'). Indexify consumes ten times more memory than Klee (row "Memory"). Indexify's increase in the run time from 0.84 seconds in the case of Klee to 10.45 seconds. This increase has two causes. The first is a very small execution times for fdlibm53. Running Indexify has an additional runtime cost because of: building the *IOT*s, changing the target type to indexes in the LLVM IR, and replacing the calls to the functions to be indexed, with their corresponding indexed versions. When the runtime to symex the initial program is big enough, the runtime reduction that our simpler constraints generate is bigger than the computational cost for Indexify. When the runtime to symex the initial program is very small, as is the case for our float benchmark, the preprocessing runtime to indexify the program is bigger than runtime reduction that the simpler constraints cause. The second is the fact that more of the indexed program's constraints are solvable allows Klee to go further down paths, increasing coverage but incurring state space explosion.

6.6 The Size of Indexified Programs

Indexification encodes the *IOT*s in the IR representation of P. Although the disk space is cheap, it is important that the size of indexified programs are manageable in practice.

We used `wc -l` to report the LOCs of the IR before and after indexification for coreutils. The total size in text lines for the initial coreutils is 1,731,007 lines; the total size for the indexified coreutils is 3,735,435. In the LLVM IR file format[8] every instruction appears on a new line. Thus, we considered the newlines in LLVM IR as a good proxy for the number of instructions in that program.

Indexification increased the total size of the IR for coreutils by 215%. Although in the worst case, the size of the *IOT* grows exponentially in the average arity of the operators, this increase is manageable in practice. We speculate two reasons for this in coreutils. The programs in coreutils do not use all the string operators in `string.h`: for example, the Unix tool `yes` does not use any string operators. Indexify adds additional code only for the *IOT*s. We construct an *IOT* only for an operator that we index. Second, Indexify encodes *IOT* as a sequence of `if` statements, one per value in the garden. LLVM translates each branch of an `if` statement into only 2 lines of IR: the predicate on one line, and the label for the true branch.

6.7 Number of Clauses

The number of clauses in queries affects the performance of symbolic execution. We compared the numbers of clauses between each original program and its indexified version. We ran Klee, and Indexify with the option: `--use-query-log solver:all,pc`. This reports all the queries sent to the underlying solver, after Klee's internal optimisations.

After the application of Indexify, symbolically executing indexed programs generated 2,920,246,627 clauses in total, while the unindexed programs generated 997,389,929. Klee on indexed programs sent 44,138,715 clauses to STP, while it sent 401,289,490 on unindexed programs.

[8]http://releases.llvm.org/2.7/docs/LangRef.html

Indexify generated three times as many constraints as Klee because of *IOT*s are encoded into queries sent to the solver. Klee's optimisations include simplifications that remove infeasible clauses from the queries, prior to sending them to the solver. These optimisations hugely benefit Indexify, removing irrelevant *IOT* entries. After the optimisations, indexed programs generates $\frac{1}{10}$ the constraints that unindexed programs do. Indexify is very effective at simplifying constraints and Klee's solver chain is very effective at dealing with the simple constraints that Indexify produces.

7 RELATED WORK

Symbolic execution (symex) binds symbols to variables during execution. When it traverses a path, it constructs a path condition that define inputs (including environmental interactions) that cause the program to take that path. Symex has some well known limitations [Cadar and Sen 2013] including path explosion, coping with external code, and out-of-theory constraints.

Harman *et al.* [Harman et al. 2004] introduced the concept of testability transformations, *source-to-source* program transformations whose goal is to improve test data generation. Following Harman *et al.*, Cadar speculated that program transformations might improve the scalability of symbolic execution in a position paper [Cadar 2015]. Indexify realizes, in theory and practice, such a program transformation, rewriting a program to allow symbolic execution to cover portions of the program's state space that current symex engines cannot efficiently reach, as we show in Section 6.2. Indexify transforms expressions in a program that produce *out-of-theory* constraints into expressions that produce *in-theory* constraints. The transformed (indexed) program underapproximates the input program's behaviour: it is precise over every component of the program's state it binds to a value in G and reifies its ignorance of component's actual value when it binds $\perp$ to that component.

7.1 Handling Unknown via Concretization

Solver unknowns bedevil Symex. One way to handle them is to resort to concrete execution. *Dynamic Symbolic Execution* (DSE) [Cadar et al. 2008] does this by lazily concretizing the subset of the symbolic state for which the solver return unknown. Concolic testing [Godefroid et al. 2005; Sen et al. 2005b; Marinescu and Cadar 2012] or *white-box fuzzing* [Godefroid et al. 2008] is an extension to DSE that initially follows the path that a concrete input executes. Further, concolic testing searches a neighbourhood around the path executed under its concrete input by negating the values of the current branch conditions from the current followed path. First concolic testing flips the closest branch to the end of execution and then, it continues to do so upwards to the entry point of the program. To flip the path condition, concolic testing uses an SMT solver.

Whenever reaching a constraint that the solver cannot solve, or for which the solver times out, the concretization methods, such as DSE and concolic testing, get a concrete value that can be either random, or obtained under different heuristics [Marinescu and Cadar 2012]. In contrast Indexify keeps the value of the unsolvable constraint symbolic, but restricted to the garden. This allows Indexify to obtain higher code coverage, by considering more paths than the concretization methods, as our direct comparison with Klee [Cadar et al. 2008] (DSE) and Zesti [Marinescu and Cadar 2012] (concolic testing) shows in Section 6.1. Additionally, Indexify does not require concrete inputs from the user, as concolic testing does: Indexify's builder automatically builds the set of values that it will consider in the analysis.

Indexify generalizes the concretization techniques: our garden G allows us to reason about a finite set of values from G, instead of a single concrete value, as in the case of the concretization methods. Like concolic testing, Indexify generates test inputs from an initial set of values. For Indexify, this initial set is the set of seeds harvested from constants; for concolic testing and white-box fuzzing, the initial set is the input test suite. Indexify and concolic testing generate these inputs differently. Concolic testing uses an SMT solver to negate the path conditions of the initial inputs; Indexify constructs its garden using repeated applications of builder functions. Our intuition is that the constants in a program are discrete points in the state space of the program: the constants appear in the predicates of the decisions points in the program. By considering these points in G, we explore the behaviour of the program on the true branch of the predicate, the discrete point in the state space of the program. For example, in the predicate `if (strstr(S,"foo"))`, we would like to consider the value `foo` for `S`. Under Indexify `foo` will certainly be considered, as Indexify will automatically harvest into its seed set. Concolic testing tends to generate inputs that follow paths in the program near one of its initial inputs. Indexify, in contrast, is not tethered to a set of initial paths, so it can cover widely varying program paths.

7.2 Constraint Encoding

During its execution, a program under Symex produces constraints in the different domains of its data types, such as constraints over strings, or floats. Before calling the solver, a Symex engine needs to encode the constraints in a language that the solver understands. Some of these constraints are difficult for the solver: the literature provide us a couple of constraint encoding mechanism to cope with the difficult kinds of constraints, in particular constraints over string, or floating point values.

Strings Quine proved that the first-order theory of string equations is undecidable [Quine 1946]. Since then different techniques have emerged implementing decision procedures over fragments of string theory. All these approaches have limitations: either they do not scale; they support a fragment of string theory that is not well-aligned with string expressions developers write; they require fixed length strings; or they require a maximum length and loop through all possible lengths up to that maximum. There are three main categories of string solvers: automata, bit-vector, and word-based.

Automata-based solvers [Veanes et al. 2010] use regular languages or context-free grammars to encode the string constraints. The idea is to construct a finite-state automata that accepts all the strings that satisfy the path conditions in a program. Building this automata requires handcrafted building algorithms for the set of string operations that we intend to support and the set of values that the program accepts as inputs [Hooimeijer and Weimer 2009]. When a new string constraint is added to the path condition, these approaches refine the automaton to not accept the strings that violate the newly added constraint. The refinement process is automatic: we remove from the set of values that the automata accepts the ones that are not valid for the new constraint. Infeasible paths construct automata that do not accept any strings. For string constraints, the automaton becomes the solver. Automata-based string solvers tend not do not combine strings with other data types [Li and Ghosh 2013], as combining string automatas with other data types and operations over them, requires handcrafted initial automatas for the particular data types and operations that the program under analysis uses in conjunction with the string operations. Yu *et al.* [Yu et al. 2009] tackles the problem of using an automaton to handle both string and integer constraints, but no other data types. Within the bounds of the values it handles, Indexify combines different data types, including strings, by automatically translating them into the theory of integer equality up to its garden, the finite set of values over it is defined.

Bit-vector based symex engines convert string constraints into the domain of bit-vectors [Ganesh and Dill 2007]. The bit-vector solvers require a maximum string length and lack scalability. Specifying a maximum length per each query that symex sends to the solver would require comprehensive annotations, so symex engines use a global maximum. In general, string length is domain-specific and specifying a maximum length for an entire program is problematic: for example, a database query might be hundreds of characters long, while a command line flag can occupy only one character. The string length restriction means that users must specify a single length for all strings in the program. When too big, this length slows the solver; when too small, it limits the analysis to small strings. Thus, these engines typically execute a program over different length strings [Ganesh and Dill 2007]. The bit-vector encoding of values causes exponential blow up in model size, 2^n where n is the length of a bit-vector, since each bit becomes a propositional variable for the underlying SAT solver and hampers scalability. For a restricted set of values, INDEXIFY encodes constraints into table lookups, not bit-vectors. Running Klee on programs transformed by INDEXIFY covers more than twice the number of branches that Klee manages on the unindexed programs (Section 6). While Klee still encodes the values in an indexed program's queries into bit-vectors, these are short bit-vector encodings of integers, not long bit-vector encodings of strings.

Word-based string solvers define a subtheory that uses rewriting rules and axioms tailored to a fragment of string theory. Word-based string solvers escape Quine's result by not handling all string expressions. CVC4 [Liang et al. 2014] and S3 [Trinh et al. 2014] support constraints over unbounded strings restricted only to length and regular expression membership operators. Z3-STR [Zheng et al. 2013] supports unbounded strings together with the concatenation, string equality, substring, replace, and length operators. INDEXIFY does not constrain the syntax of string expressions; instead, it constrains their values. To do so, it finitizes string operators, restricting their definition to a finite portion of their domain and range. As Section 4 describes, INDEXIFY converts its finitized string expressions into constraints over integer equality, then forwards them to an SMT solver.

Floating Point Bit-blasting is the most widely used technique for solving floating-point constraints. Bit-blasting converts floats into bit-vectors and encodes floating point operations into formulae over these bit-bectors. Bit-blasting floating-point constraints generates formulaes that require a huge number of variables. For example, Brillout *et al.* [Brillout et al. 2009] showed than when using a precision of only 5 (mantissa width) addition or subtraction over floating point variables requires a total of 1035 propositional variables to be encoded in bit-vector theory. In a similar scenario, multiplication or division requires a total of 1048 propositional variables. Because of this, bit-blasting for floating point constraints does not scale to large programs [Darulova and Kuncak 2014].

Testing is also used to solve floating point constraints. Microsoft's Pex symex engine can use the FloPSy [Lakhotia et al. 2010] floating point solver. FloPSy transforms floating point (in)equalities into objective functions. For example, the predicate `if((Math.Log(a)== b)` becomes the objective function $|Math.Log(a) - b|$ for which we want to find values of a and b that yield 0. FloPSy uses hill climbing [Clarke 1976] to find values for a and b. Fu *et al.* proposes Mathematical Execution (ME) [Fu and Su 2016]. They capture the testing objective through a function that is minimised via mathematical optimisation to achieve the testing objective. Given a program under test P, they derive another program P_R called representing function. P_R represents how far an input $x \in \text{dom}(P)$ is from reaching the set $x|P(x)$ is wrong. P_R returns non-negative results and P_R is the distance between the current input and an error triggering input. Further, they minimise P_R. Souza *et al.* [Souza et al. 2011] integrate CORAL, a meta-heuristic solver designed for mathematical constraints, into PathFinder [Visser et al. 2004].

Klee-FP [Collingbourne et al. 2011] and Klee-CL [Collingbourne et al. 2014] replaces floating-point instructions with uninterpreted functions. Klee-CL and Klee-FP apply a set of canonizing rewritings and further do a syntactic match of floating-point expressions trees. Both Klee-CL and Klee-FP solve floating point constraints by proving that them are equivalent (or not) with an integer only version of the program. They do this by matching the equivalent expression trees. Thus, the main limitation is the requirement of having the both version of the programs available: the floating point and the integer implementation. INDEXIFY does not require any existing integer implementation.

Other approaches use constraint programming [Michel et al. 2001] to soundly remove intervals of float numbers that cannot make the path condition true. Botella *et al.* [Botella et al. 2006] solve floating point constraints using interval propagation. Interval propagation tries to contract the interval domains of float variables without removing any value that might make the constraint true. These approaches are either imprecise or do not scale. INDEXIFY translates floating point arithmetic into equality theory over integers that is solvable in polynomial time [Barrett et al. 2009].

7.3 The State Of the Art in Symbolic Execution

Significant research has tackled the problem of applying symex to real world programs [Cadar and Sen 2013]. INDEXIFY complements this work: INDEXIFY is a program transformation that is applied prior to symex; and thus can be applied in conjunction with any technique that improves symex.

Some approaches aim to improve a symex engine's interactions with the constraint solver. Klee [Cadar et al. 2008] and Green [Visser et al. 2012] cache solved constraints. Lazy initialization delays the concretization of symbolic memory thereby avoiding the concretization of states that additional constraints make infeasible [Rosner et al. 2015]. Memoized symex [Yang et al. 2012] and directed incremental symbolic execution [Person et al. 2011] reuse query results across different symex runs.

Other approaches improve the scalability of Symex by mitigating the path explosion problem. Veritesting /citeavgerinos2014enhancing proposes a path merging technique that reduces the number of paths being considered as a result of reasoning about the multiple merged paths simultaneously. MultiSE [Sen et al. 2015] perform symbolic execution per method, rather than per the entire input program. MultiSE merges different symex paths into a value summary, or a conditional execution state. Further, we can symex each method in a program starting from its value summary, rather than from the program's entry point.

For an unsolvable constraint, symex concretizes its variables, causing incompleteness: it cannot reason on the rest of the state space. Pasareanu *et al.* [Păsăreanu et al. 2011] delay concretization to limit the incompleteness. They divide the clauses into simple and complex ones. When a simple clause is unsatisfiable, the entire PC becomes unsatisfiable. This can avoid reasoning about the complex clauses. Khurshid *et al.* [Khurshid et al. 2003] concretizes objects only when they need to access them.

A recent study [Dong et al. 2015] show that 33 optimization flags in `LLVM` decrease symex's coverage on `coreutils`. Overify [Wagner et al. 2013] proposes a set of compiler optimisations to speed symex. Sharma *et al.* [Sharma 2014] exploit these results and introduce undefined behaviour to trigger various compiler optimisations that speed up symex [Cadar 2015]. Under-Constrained Symex [Ramos and Engler 2015] operates on each function in a program individually. Abstract subsumption [Anand et al. 2006] checks for symbolic states that subsume other ones and remove the subsumed ones. Ariadne transforms numerical programs to explicitly check exception triggering conditions in the case of floats [Barr et al. 2013]. As INDEXIFY is a program transformation step prior to symex, we can take advantage of these optimisations by running INDEXIFY in conjunction with them.

8 CONCLUSION

We introduced indexification, a novel technique that rewrites a program to constrain its behaviour to a subset of its original state space. Over this restricted space, the rewritten program under-approximates the original; its symbolic execution generates tractable constraints. We realized indexification in INDEXIFY and show that it automatically harvests program constants to define restricted space permitting symex to explore paths and find bugs that other symbolic execution techniques do not reach.

REFERENCES

Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2006. Symbolic execution with abstract subsumption checking. In *International SPIN Workshop on Model Checking of Software*. Springer, 163–181.

Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 549–560.

Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. *Handbook of satisfiability* 185 (2009), 825–885.

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 307–321.

Bernard Botella, Arnaud Gotlieb, and Claude Michel. 2006. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability* 16, 2 (2006), 97–121.

Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. IEEE, 69–76.

Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 906–909.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. 2013. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems* 29, 7 (2013), 1758–1773.

Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* 3 (1976), 215–222.

Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2011. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the sixth conference on Computer systems*. ACM, 315–328.

Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2014. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering* 40, 7 (2014), 710–737.

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. *Acm Sigplan Notices* 49, 1 (2014), 235–248.

LLVM Developers. 2016a. LLVM. http://llvm.org/. (2016). Accessed: 2016-07-02.

ReactOS Developers. 2016b. ReactOS. http://www.reactos.org/. (2016). Accessed: 2016-07-02.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *International Static Analysis Symposium*. Springer, 236–252.

Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the influence of standard compiler optimizations on symbolic execution. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 205–215.

Zhoulai Fu and Zhendong Su. 2016. Mathematical Execution: A Unified Approach for Testing Numerical Code. *arXiv preprint arXiv:1610.01133* (2016).

Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*. Springer, 519–531.

Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis (ISSTA 2013)*, Mauro Pezzè and Mark Harman (Eds.). ACM, Lugano, Switzerland, 302–313.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI 2005)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223.

Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

Pieter Hooimeijer and Westley Weimer. 2009. A decision procedure for subset constraints over regular languages. *ACM Sigplan Notices* 44, 6 (2009), 188–198.

Susmit Jha, Sanjit A Seshia, and Rhishikesh Limaye. 2009. On the computational complexity of satisfiability solving for string theories. *arXiv preprint arXiv:0903.2825* (2009).

Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 553–568.

Stephen Cole Kleene. 1951. *Representation of events in nerve nets and finite automata*. Technical Report. DTIC Document.

Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

Kiran Lakhotia, Phil McMinn, and Mark Harman. 2009. Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?. In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*. Windsor, UK, 95–104.

Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. 2010. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*. Springer, 142–157.

Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conference*. Springer, 15–31.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL (T) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*. Springer, 646–662.

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

Paul Dan Marinescu and Cristian Cadar. 2012. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 716–726.

Claude Michel, Michel Rueher, and Yahia Lebbah. 2001. Solving constraints over floating-point numbers. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 524–538.

Corina S Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 34–44.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 504–515.

Willard V Quine. 1946. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic* 11, 04 (1946), 105–114.

David A Ramos and Dawson R Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code.. In *USENIX Security*. 49–64.

Barry K Rosen. 1973. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM (JACM)* 20, 1 (1973), 160–187.

Nicolás Rosner, Jaco Geldenhuys, Nazareno M Aguirre, Willem Visser, and Marcelo F Frias. 2015. BLISS: Improved symbolic execution by bounded lazy initialization with sat support. *IEEE Transactions on Software Engineering* 41, 7 (2015), 639–660.

Koushik Sen, Darko Marinov, and Gul Agha. 2005a. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.

Koushik Sen, Darko Marinov, and Gul Agha. 2005b. CUTE: a concolic unit testing engine for C. In 10^{th} *European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*, Michel Wermelinger and Harald Gall (Eds.). ACM, 263–272.

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 842–853.

Asankhaya Sharma. 2014. Exploiting undefined behaviors for efficient symbolic execution. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 727–729.

Matheus Souza, Mateus Borges, Marcelo d?Amorim, and Corina S Păsăreanu. 2011. CORAL: solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*. Springer, 359–374.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1232–1243.

Margus Veanes, Nikolaj Bjørner, and Leonardo De Moura. 2010. Symbolic automata constraint solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 640–654.

Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 58.

Willem Visser, Corina S P?s?reanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 97–107.

David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.. In *NDSS*. 2000–02.

Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. -Overify: Optimizing Programs for Fast Verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 144–154.

Fang Yu, Tevfik Bultan, and Oscar H Ibarra. 2009. Symbolic string verification: Combining string analysis and size analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 322–336.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 114–124.

Appendix C

Retype: Changing Types to Change Behaviour

Retype: Changing Types to Change Behaviour

Abstract

To retype a program is to change the type annotations of some of its variables and then replace the operators used in operations on those variables so that the new program type checks. Retyping allows developers to change the width of a variable, *e.g.*, `float` $\rightarrow$ `double` to trade performance against numerical stability, or associate additional information with it, like taint. RETYPE is the first tool to automate retyping an arbitrary part of a program, such as a function or a single variable. Retyping a region can introduce function overloads, as when an operator given a new definition within a region, while still using its original definition elsewhere. To handle overloading, RETYPE employs intersection types, escaping their undecideability by only retyping programs that type check prior to retyping and are therefore in β-normal form. RETYPE rewrites type contexts to produce constraints in 2-SAT whose solution either is a valid type assignment or a detailed error message. The new target type must be a subtype of the type it replaces; if it further obeys the Liskov substitution principle, we show that the retyped program is semantics-preserving with respect to the original program's behaviour. We have realised RETYPE in a practical, scalable tool which successfully retyped Pidgin (363K LOC). We demonstrate its utility by semi-automating bespoke examples of retyping operations from recent research, including a taint analysis and an energy consumption optimiser.

1. Introduction

Type systems rule out incorrect behaviours, document code, support code completion, and facilitate code search and navigation. These uses cut across the development process, improving and enabling new feature development, optimisation, and maintenance. Using types to introduce controlled changes to a program's behaviour is, however, less explored [42–44].

To retype a program is to change the type annotations of some of its variables and then replace the operators used in operations on those variables so that the new program type checks. Retyping has two main use cases: firstly, it replaces operators with semantically equivalent ones for refactoring, aiming to preserve the semantics of the input program [42–44]; secondly, it introduces new behaviour to accommodate some analysis or optimisation techniques, leaving the program's control flow outside the replaced operators unchanged. The new behaviour type checks, by construction. For example, retyping allows a developer to change the width of a numerical type to trade performance against numerical stability, inject instrumentation as to track taint, or reduce state space of the program to make a subset of its behaviour amenable to automated reasoning. To elaborate the first example, developers may change a numerical variable's type to increase its width to avoid integer overflow or to increase precision of floating point numbers; conversely, they may decrease a variable's width to use bits more efficiently and improve performance.

Although retyping is currently mostly manual, it is so tedious that it is not uncommon to find tools, purpose-built for special cases of retyping, usually for a fixed type conversion. Indeed, retyping occurs often enough and is sufficiently error-prone that two popular IDEs support limited retyping: Eclipse and IntelliJ. Both IDEs use interprocedural dataflow to identify the uses of a variable being retyped. These tools only retype a single variable at a time and address only the problem of identifying the uses of a retyped variable, leaving the replacement of operators and the handling of constants and externals to the developer. In the research literature, Balaban *et al.* [1] introduced ReplaceClass, which uses type constraints to automatically change a class to a semantically equivalent one. ReplaceClass is restricted to Object-Oriented languages and globally replaces uses of one class with another throughout a program. While ReplaceClass aims at preserving the semantics of the input program, RETYPE does not do so. RETYPE can be both semantic preserving, or not, according to the user provided definitions for the new data type.

RETYPE supports retyping portions of a program and automatically handles external functions like library and system calls. For example, a developer might want to perform taint analysis for only variables in the security critical parts of the input program, rather than in the entire program. The program regions whose type annotations we change are retypeable, and the remainder are not (Sec. 3.1). Values that flows into or out of a retypeable region must be converted. For this, RETYPE uses casts, as detailed in Sec. 3.3. If an unretypeable region uses a operator that is retyped, that region will retain the operator's original definition. Therefore, RETYPE introduce overloadings, by preserving the old operator definitions.

Handling operator overloading is challenging. One can use intersection [12] ($\cap$) types or fresh names for this. Fresh names generate constraints for each overload of a function; while $\cap$-types generate one constraint for all the overloads. $\cap$-types are a well-studied topic in type theory, naturally and elegantly handle overloading by allowing operators (function definitions) to have multiple types. However, inference over $\cap$-types is undecidable in general [29]. Thus, the only practical use of $\cap$-types to date is FaceBook's Flow, which requires manual annotation. A program that type checks against their base language's type system, which does not use $\cap$-types, are in β-normal form. Our insight is that we can escape this undecidability result by inferring $\cap$-types only over programs that type check in the base language's type system, essentially putting two type systems in sequence (Sec. 3.2).

This paper implements a retyping tool, called RETYPE. This tool takes an input program, an old type, and a new type, a subtype of the old type, and automatically replaces the old type with the new one, updating operator definitions and applications as needed. RETYPE infers the operators that are required and provides this information to the user, if the user does not supply all needed operators. A core novelty of RETYPE is that it can replace a type with a subtype in languages that do not support subtyping, like C. If RETYPE reports no error, the resulting program type-checks by construction.

RETYPE uses a term rewriting system to rewrite type bindings. This term rewriting system operates on ground terms, which avoids type variables, and therefore is decidable [11]. The resulting type bindings express RETYPE's type inference problem as 2-SAT formula, Sec. 3.4, in contrast to ReplaceClass which must resort to backtracking. Thus, we reduce type inference in our system to constraint solving, for which we use Z3 [38], an off-the-shelf SMT solver. We use constraints, instead of unification, for type inference so we can give precise error messages that include the location of an operator application that RETYPE could not resolve.

RETYPE was successfully tested on popular real world programs with significant sizes, such as `Pidgin` [16] (363K LOC).

The main contributions of this paper follow:

1. The formalisation of retyping, comprising a sound type assignment system and a confluent and terminating term rewriting system over type contexts (Sec. 3);
2. The rewriting of type contexts to produce constraints in 2-SAT which is solvable in polynomial type (Sec. 3); and
3. The realisation of RETYPE (Sec. 4) in a tool and a demonstration of its utility on real world examples (Sec. 5).

Upon acceptance of this work, we will publish RETYPE at `www.utopia.com`.

2. Motivating Example

When a developer changes `float` to `double` to increase the accuracy of their results, they are retyping their programs. When a developer replaces `signed int` with `unsigned int` to prevent the exception of array index out of bound, they are retyping their programs. Indeed, bespoke attempts to semi-automate are common. Ariadne [6], a symbolic execution engine that automatically finds exception-triggering inputs for floating-point programs, extensively uses a tailored, float to arbitrary precision rational, retype transformation. μscalpel [5, 31] automatically transplants code, an 'organ', from one code base to another; it uses a custom retype operation to transform the organ to use types in the target codebase. Upon deciding the most suitable Floating-Point precision for a given problem, programmers often face a trade-off between accuracy, and performance and energy consumption. In this situation, RETYPE automates the tedious and error-prone type replacement, and helps programmers quickly approach the sweet spot.

Taint analysis [33] marks (taints) values from certain sources and tracks the propagation of these markings, usually checking whether the values reach some sink. It shines in software security, but also shows its strength in other areas, such as testing and debugging. The developer has to identify the variables of interest as *retypeable* terms for RETYPE. These can be variable-granular or arbitrary subsequences of the program that contains a sequence from the source to the sinks. RETYPE replaces the initial type of retypeable terms with one that wraps the type with taint flags and operations that update the taint, as shown in Sec. 5.2. The retyped program automatically handles explicit information flow (data dependence), because its control flow, outside of the replaced operators, is unchanged. To enable taint analysis for a program of multiple types, the user can simply apply RETYPE multiple times.

A user wishing to use RETYPE to inject taint tracking types would have to supply retype with pointer types that include a taint field and corresponding operators, like dereference. RETYPE is a general tool for changing types. A specific usage as taint analysis requires some additional human work.

Fig. 1c contains a taint analysis example adapted from a famous paper on dynamic taint analysis which introduces a general framework for previous ad-hoc techniques and considers both control and data flow [10, Section 2, Figure 1]. Suppose we want to replace the primitive data type `int` with the ADT `TaintedInt`. Changing only the type annotations will result in compilation errors for three reasons. First, operators, such as `+` and `*`, are not defined on `TaintedInt` in the original program. Second, the types of literal constants, such as `11` and `5`, cannot be changed. Last but not least, the definitions

```c
int foo(){
    int i = 3;
    return i * i * i;
}
```

(a) A not-to-be retyped function.

```c
typedef struct {
    int value;
    bool tainted;
} TaintedInt;
```

(b) The new, target type.

```c
int main(){
    int a,b,w,x,y,z;
    a = 11;
    b = 5;
    scanf("%f", &b);
    w = a + foo();
    w = a * 2;
    x = b * 1;
    y = w + 1;
    z = x + y;
}
```

(c) P, the code to retype.

Figure 1: Input code for RETYPE's motivating example; the gray background indicates that function `foo()` is unretypeable; Fig. 1b defines `TaintedInt` the target type for RETYPE.

```c
TaintedInt op+(TaintedInt op1, TaintedInt   op2){
    TaintedInt result = {op1.value + op2.value,
        op1.tainted || op2.tainted};
    return result;
}
TaintedInt constructTaintedInt(int intVar){
    TaintedInt result = {intVar, false};
    return result;
}
```

Figure 2: Illustrative example of the user provided operators. In practice, RETYPE requires these definition and more besides (*e.g.* assignment operator).

of external functions that are not retypeable, such as `scanf()` and `foo()`, cannot be changed, detailed in Sec. 3.1. If RETYPE encounters a variable of the initial type that is passed to an external function by reference, such as variable `b` in the example, it always constructs a free variable of the new type to temporarily store the value. This free variable updates its value in the function and will be cast back after the function execution. RETYPE casts the free variable back after the function execution. If a variable is passed to an external function by value, RETYPE adds an explicit cast to reconcile the conflict. Sec. 3.3 discusses the cast in details. RETYPE copes with external functions' return values and literal constants similarly.

RETYPE requires the user to provide the missing operator definitions involving the new type. If the names of the operators over the old and new type differ, the user needs to manually map each definition to the actual operator in the program. RETYPE applies operator replacement automatically. Fig. 2 shows an example of the user defined operators for `TaintedInt`. RETYPE receives a definition for `op+` that takes two `TaintedInt` values, and returns a `TaintedInt` value, as well as a cast operator. The last one takes an `int` value, and converts it to a `TaintedInt` value. Fig. 3 shows the retyped program, under the inputs in Fig. 2. Note that RETYPE is flexible in its acceptance of operator definitions, as long as the retyped program type checks and compiles.

Without the required definitions, RETYPE warns the user with an error message that specifies which operators are missing. Below is a typical error message, when the user does not provide the definition of `+` over the new type `TaintedInt`:

```
ERROR: "+", application "1" not handled!
APPLICATION: w + 1
LOCATION: taint.c, line 9
```

```c
int[1] main(){
    TaintedInt a, b, w, x, y, z;
    a = constructTaintedInt(11);
    b = constructTaintedInt(5);
    int freeVar = constructInt(b);
    scanf("%f", &freeVar);
    b = constructTaintedInt(freeVar);
    w = a + foo();
    w = a * constructTaintedInt(2);
    x = b * constructTaintedInt(1);
    y = w + constructTaintedInt(1);
    z = x + y;
}
```

Figure 3: The retyped program.

3. Formalism

Algorithm 1 defines RETYPE's highlevel algorithm. RETYPE takes as input P the program to retype, τ_o the "old" type to replace, and τ_n the "new" type with which to replace τ_o. RETYPE also takes two optional functions: r that specifies which terms in P are *retypeable* and m that maps methods of τ_o to ones of τ_n. The function r defaults to all terms internal to P; m defaults to the identity function. The new type τ_n must be a subtype of τ_o with respect to τ_o's retypable uses in P. In the case of non-OOP languages, τ_o and τ_n share definitions: they are not encapsulated. Thus, we do not introduce classes in non-OOP languages. Both τ_o and τ_n denote both a type name and its definition; we specify which when it is not clear from context.

First, Algorithm 1 marks retypeable terms in P (line 1), because we may wish to selectively retype variables or parts of a program. Sec. 3.1 describes the details. For the retypeable terms, line 2 uses m builds Γ, a type context, augmented with term locations, in the base language that binds retypeable terms to both τ_o and τ_n, as we detail in Sec. 3.2. On line 3, Algorithm 1 rewrites Γ into RETYPE's type context, then uses it to rewrite P into P', which uses τ_n (Sec. 3.3). Finally, Algorithm 1 type checks P' (Sec. 3.4), which may fail because τ_n does not define an operator for a retypeable call or because more than one operator matches a call. When type checking fails, RETYPE reports the error and the location of the problematic call. Otherwise, it returns the retyped program. Note that in this section we just sketch the proof ideas. Please refer to the technical report [32] for the detailed proofs.

3.1 Retypeability

When retyping a program, developers will often want to control the specification of retypeable terms, perhaps to a single function or even to the granularity of a single variable. A program is sequence of terms, in the sense of lambda calculus [21], that are either internal and mutable, or external and immutable, like constants, third party library or system calls. To control which terms are retyped, RETYPE allows its user to specify r, an operator that marks some internal terms as *retypeable*, implicitly externalizing the rest. A term can be

[1] In C, the `main` function is unretypeable because it connects the program to C's run-time system and is therefore external.

Algorithm 1 RETYPE$(P, \tau_o, \tau_n, [r,][m])$

```
inputs $P$, the input program
        $\tau_o$, the old type
        $\tau_n$, the new type
        $r$, a function that identifies retypeable terms
        $m$ maps operator names in $\tau_o$ to $\tau_n$
    The function $r$ defaults to all terms internal to $P$ and $m$
    defaults to identity.
1: $P^r := r(P)$                                   # Sec. 3.1
2: $\Gamma :=$ infer$(P^r, \tau_o, \tau_n, m)$     # Sec. 3.2
3: $P' = \Phi(P, \Gamma)$                          # Sec. 3.3
4: return $P'$ if $tc(P')$                         # Sec. 3.4
```

external by definition (implicit) or because r designates it external (implicit). By default, all internal terms are retypeable.

A subtle application of r arises when not all terms in a statement are retypeable. For example, assume variable `a` is retypeable in `a = 3;`. One might expect to use a cast on `3`, when `a` is retyped. However, `3` is not retypeable and we cannot rewrite it. Thus, for RETYPE to succeed, the user has to provide a new definition of the assignment operator `=`, which takes as input an `int` variable and returns a value of the new type of `a`; otherwise RETYPE errors. If both `a` and `3` are retypeable, then RETYPE uses a cast in the absence of a corresponding assignment operator. The key technical challenge in RETYPE is to identify which operator/function application to replace, and with what. RETYPE solves this problem, as we show in Sec. 3.

Fig. 1c in Sec. 2 shows both kinds of externality. Line 5 calls `scanf()`, an external function that is defined in the system libraries and therefore not retypeable. Line 6 calls the function `foo()`. Although this function is defined in Fig. 1a, the user explicitly declared it unretypeable, as the gray background indicates. Thus, `scanf()` and `foo()` are unretypeable.

3.2 RETYPE's Type Inference

RETYPE requires type inference. Once we change the type of a variable, we need to infer the types of calls to operands of the replaced type, for not introducing any type errors in a program. For example, if in Fig. 1c we retype `int` to `TaintedInt`, without inferring the types of calls to operator `+`, we wouldn't see that on line 7 the initial operator `+` does not accept an `int` as a first argument.

If all the terms in P are retypeable, RETYPE would not introduce overloading. However, this condition is too strong, because of implicit externality, *e.g.*, we cannot change the definitions of a system call or the type of a constant. Retyping parts of a program inevitably results in operator overloading, because an operator involving τ_o to be retyped is given a new definition, but its old definition might still be used outside the retypeable region. Therefore, we have to retain an operator's old definition, along with the new one, to preserve type correctness and the original behaviour, thereby introducing overloading. For example, at line $9:$ `y = w + 1;` and $10:$ `z = x + y;` of our running example (Fig. 1c), the operator '`+`' has two call sites. The first call has an unretypeable operand, the constant '`1`', and both operands are retypeable in the second one. Under these conditions, RETYPE infers two different types for `+`, `TaintedInt` $\rightarrow$ `int` $\rightarrow$ `TaintedInt` for the first application and `operator+`: `TaintedInt` $\rightarrow$ `TaintedInt` $\rightarrow$ `TaintedInt` for the second.

Intersection type systems, or CDV from Coppo-Dezani-Veni, can type all terms that have a head normal form [12], *i.e.* they have no β redex in their head position, including operator overloading, but are undecidable, because $\cap$-types assign the same type to β–equivalent terms, but the β–equivalence of two terms is undecidable [29]. When retyping, we assume that the program type checks under the base language's type system, and infer from it the types of atomic terms. These terms are in β–normal form, otherwise the input program would not have type checked. Thus, retyping avoids β–reductions and escapes the undecidability of CDV. RETYPE needs only to perform type inference for the $\rightarrow$ constructor. These inferences are decidable [7, 9].

RETYPE expresses the type of `operator+` discussed above as: (`TaintedInt` $\rightarrow$ `int` $\rightarrow$ `TaintedInt`) $\wedge$ (`TaintedInt` $\wedge$ `TaintedInt` $\wedge$ `TaintedInt`) , or: `TaintedInt` $\rightarrow$ `(int` $\cap$ `TaintedInt)` $\rightarrow$ `TaintedInt`. These two formulas are equivalent because $\wedge$ distributes over $\rightarrow$. If τ_n does not define the required overloadings, these lines are unchanged in the retyped program. Otherwise, RETYPE casts the operands as a last resort, as discussed in Sec. 3.3.

We now define a type context, and illustrate how RETYPE generates the type context for the example in Fig. 1c.

Definition 1. A **type context** Γ is any finite set of pairs, comprehending type assignments (TA) and locations: $\Gamma = \{(x_1 : \tau_1, l_1), \cdots, (x_n : \tau_n, l_n)\}$. The TA's subjects are term-variables.

A type context contains definitions and function calls. Definitions naturally include function and variable definitions. To rewrite type annotations and calls in line 3 in Algorithm 1, we adapted the standard definition of the type context [21] to include the location of the term, using path, file, line number as our coordinates, as l_i. We use the location information when we produce the retyped program (Sec. 3.4).

To support retyping, RETYPE infers the type context Γ across the retypeable terms in the input program and τ_n. The map m (Algorithm 1) connects terms in the input program P to their replacement in τ_n. m is an optional argument, that is not required when 1) the user-supplied operators have the same name as the operators they will replace in operation over operands whose types have changed and 2) P is written in a language that supports signature-based dispatch. If these assumptions do not hold, we use the m to connect terms in the input program P to their replacement in τ_n, when building Γ.

In our motivating example (Fig. 1c), we denote the original type `int` with τ_o, and the target type `taintedInt` with τ_n. We identify all the usages of variables and operators involving τ_o in the input program. We assign unique terms to each operators, such as $x_2y_2z_2$ (*i.e.* two variables for the two arguments that `scanf` accepts, and one for its return) for `scanf` and $x_4y_4z_4$ for `+`. This prevents overwriting, as might happen when two local variables are retypeable, have the same name, but have different types. After applying α-renaming (bound variables) and substitution (free variables) to the previously identified λ-terms, we obtain the operators' types. For brevity, we only show the type of `+` below and use it throughout the remainder of this section.

For clarity, we replace RETYPE's internal names with meaningful ones. For `+`, its two arguments are op_1 and op_2, and its body is $body+$. The inferred types of operator `+` from its uses in the example are then: `6: a + foo()`: $(\tau_o \to \tau_o \to \tau_o)$; `9: w + 1` $(\tau_o \to \tau_o \to \tau_o)$; `10: x + y` $(\tau_o \to \tau_o \to \tau_o)$.

From the types of the operators, we infer the types of the atomic terms, then construct Γ. Fig. 2 defines τ_n's operators. First, we eliminate the redundant type assignments. For example, the code segment `a+b; b+c;`, where `a`, `b`, and `c` are retypeable, generates duplicate type assignments for `+`'s first operand. Next, we group type assignments of the same atomic term. In our type contexts, r marks all the retypeable terms and $\overline{r}$ marks all the unretypeable terms. The type context is: $\Gamma = \{\{op_1 : \tau_o^r, op_1 : \tau_n\}, \{op_2 : \tau_o^{\overline{r}}, op_2 : \tau_o^r, op_2 : \tau_n\}, \{body+ : \tau_o^r, body+ : \tau_n\}\}$.

Traditionally, base languages type systems do not use $\cap$, because of its undecidability. However, we need it to handle function overloading, as we showed at the start of this subsection. $\uplus$ denotes disjunctive union; RETYPE's TA uses it to handle unions [4]. We use $\dashv$ to denote *casts*. RETYPE's TA uses casts to convert terms that flow into and out of a retyped region. Sec. 3.3 details $\uplus$ and $\dashv$. Now, we define a TA for RETYPE that meets all these requirements:

Definition 2 (RETYPE Type Assignment)**.** $TA(\uplus, \cap, \dashv, \omega)$ is defined by the set of types defined by the following grammar:

$$\begin{aligned} \phi, \psi &::= \varphi \mid \omega \mid \sigma \to \psi & \\ \gamma, \delta &::= \phi_1 \uplus \cdots \uplus \phi_n & (n \geq 1) \\ \sigma, \tau &::= \gamma_1 \cap \cdots \cap \gamma_n & (n \geq 1) \end{aligned}$$

In $TA(\uplus, \cap, \dashv, \omega)$ every type can be written as: $\sigma_1 \to \ldots \sigma_n \to \phi$, where ϕ is a type-variable or ω. We write $\Gamma \vdash m : \sigma$ for statements derivable under TA.

To type check arbitrary terms in the typed λ-calculus, we need to rewrite them to a normal form under β-reduction. Whether or not two terms are equal under β-reduction is β-equivalence.

Theorem 1 (β-equivalence for $TA(\uplus, \cap, \dashv, \omega)$)**.** $X_1 =_\beta X_2 \wedge \Gamma \vdash X_1 : \sigma \Rightarrow \Gamma \vdash X_2 : \sigma$

Proof Idea. Our proofs must account for the fact that TA incorporates union types. Barbanera *et al.* [3] proved that an intersection type assignment system with union type is equivalent with the same type assignment system without union type. Thus, to prove properties of $TA(\uplus, \cap, \dashv, \omega)$, we need to prove them to hold over our type assignment system without union types and provide a suitable translation between the two type assignment systems.

Let $TA_{\neg\uplus}$ be our type system without $\uplus$. β equality for $TA_{\neg\uplus}$ can be proved by induction over distinct redexes. Next we define the following translations: $\sigma \backsim (\tau_o \uplus \tau_n)$; $\sigma_o \backsim \tau_o$; $\sigma_n \backsim \tau_n$; Under this map, any resolution of the disjunctive union type operator can be replaced by using the $\cap$-type. The two possible resolutions are τ_o or τ_n. By intersecting σ_o with σ, and σ_n with σ we obtain τ_o and τ_n respectively. Therefore, the β equality for $TA(\uplus, \cap, \dashv, \omega)$ holds. □

The function *infer* at line 2 Algorithm 1 realises the inference process described in this subsection: it infers types over $TA(\uplus, \cap, \dashv, \omega)$, and builds the type context Γ over P and τ_n.

3.3 RETYPE's Type Context Rewriting

Φ is RETYPE's term rewriting system that rewrites a type context Γ. To output the retyped program, Φ tracks the changes it makes in the type context, transforming Γ into a set of triples: initial term, modified term, and their location. RETYPE leverages the fact that, of the myriad of types, Φ's rewriting rules need to consider only two, constant types: τ_o and τ_n. Thus, Φ is a ground term rewriting system over two constants.

Terms of type τ_n that flow out of a region undergoing retyping must be converted; terms of type τ_o that flow into that region must also be converted if they are used in a operation whose operator is retyped, *e.g.*: `6 : w = a + foo();` External functions like library and system calls are one source of these terms; regions the developer decided not to retype are the other. For all such values, τ_n must define casts, which convert old values into new ones in-bound into a retyped region and extract old values from the new type outbound. Retype must inject these casts at the appropriate points. Note that casts are, in general, Turing-complete, not merely a substitution operator on type annotations. Consider for example extracting values from an ADT in a downcast in `C++`. To model these casts, we define a swap function over type annotations: $\dashv(x)$ returns τ_o if $x = \tau_n$, and τ_n otherwise.

We restrict our term rewriting system only to retypeable terms, as returned by r (Sec. 3.1). Casts complicate rewriting. We must first identify and mark retypeable terms to produce an intermediate type context over which we then match τ_n's operators, before, as a last resort, applying the casts. If we tried to perform these rewritings simultaneously, the resulting term rewriting system would not be left linear. Thus, Φ rewrites Γ, the type context of the input program P, in two phases. First, it replaces τ_o with the union type $\tau_o \uplus \tau_n$ for all retypeable terms t, as determined by $r(t)$. This phase produces Γ_1', the rewritten type context, by applying the following rule: $R_1 = \tau_o \longrightarrow (\tau_n \uplus \tau_o)$.

The second phase handles externals, such as constants and external library calls. RETYPE uses R_2 to apply casts to terms at the boundary between retypeable and unretypeable terms, only when a more precise user-supplied operator is not available. Given the term t, the $external?(t)$ selects constants and external calls, but does not match user-supplied operators. The second rule rewrites Γ'_1, producing Γ', the output of Φ: $R_2 = \tau_i \longrightarrow \dashv\!\exists(\tau_i)$ if $external?(t) \wedge \tau_i \in \{\tau_o, \tau_n\}$.

RETYPE separates its rewriting into two phases to ensure that the casts are not applied indiscriminately. The first phase replaces τ_o in retypeable terms. If this replacement were unilaterally to τ_n, then the user would not be able to include operators in τ_n whose signatures include τ_o. Applying R_1 on τ_n is a no-op.

We want to retype Fig. 1c from `int` to `TaintInt`. However, the constant `1` at line `9` is unretypeable. Without a union type, the user must provide two different functions to replace the initial definition of `+`: `TaintInt` $\rightarrow$ `int` $\rightarrow$ `TaintInt` (f_1) and `TaintInt` $\rightarrow$ `TaintInt` $\rightarrow$ `TaintInt` (f_2). f_1 can capture both uses; when this is the desired outcome, f_2 is redundant. Union types address this problem, by reducing the number of operators the user must supply. Union types assign one of several types (as opposed to $\cap$-types, which define types that combine a set of types) to a variable, including operands/function parameters. Our use of union types allows Φ to rewrite τ_o into the disjunctive union $\tau_n \uplus \tau_o$, thereby deferring the resolution of the exact type to use until the operator matching phase, allowing f_1 to be used on both lines `9` and `10` in our running example that become:`9 y = w + 1;` , and `10 z = x + constructInt(y);`

We now formally define Φ, its term rewriting system, and sketch the proofs for its decidability and, since we are applying it to type contexts, its soundness.

Definition 3 (RETYPE Term Algebra)**.** RETYPE Term Algebra is a freely generated algebraic structure over the following signature: $\sigma = (\{:\}, \{\dashv\!\exists\}, ar)$, where ':' function symbol refers to the type assignment expressions; $\dashv\!\exists$ is the cast operator; and 'ar' function assigns the arity to every function or relational symbol: ':', and '$\dashv\!\exists$'. The set of finitary operations defined on this algebraic structure is the set of type constants (nullary operators) that we use: $\{\tau_o, \tau_n\}$.

Definition 4 (The RETYPE Term Rewriting System)**.** $\Phi = R_1 \circ R_2$ is the rewriting system over RETYPE Term Algebra (Definition 3) that composes the two rewrite rules R_1 and R_2.

Theorem 2 (Decidability)**.** Φ is decidable.

Proof Idea. To show that Φ is decidable, we show that Φ is confluent and terminating (strongly normalisable) [25]. Confluence assures that each term has a unique normal form, independent on the order of rewritings; while the terminating property ensures that the number of rewritings to reach a term's normal form of a term is finite.

To show that a system is confluent, it is enough to show that it is regular, that it is both left-linear and non-overlapping. A term rewriting system is *left-linear* if and only if all variables appear only once on the left hand side of a rule. A system is *non-overlapping* if and only if there is only one way to reduce a redex within a term. Φ is left-linear because R_1 and R_2 are trivially left-linear and applied separately; Φ is non-overlapping because only one reduction rule can be used for reducing a redex at a time. Therefore, Φ is confluent. Φ terminates because it contains only ground terms and termination is known to be decidable in ground rewriting systems. $\square$

Theorem 3 (Soundness)**.** Φ is sound.

Proof Idea. A type system is sound if and only if the execution of a well-typed program is free of type errors. As is conventional, the proof of soundness rests on proving that the rewritten program P' preserves types and progresses [13, 20]. It is reasonable to assume that the input program P is produced by a compiler whose type system is sound. Thus, P preserves types and progresses. We also assume the user-provided operators are correct.

The preservation of P' means that when P' is evaluated, every step preserves the type modulo τ_n. Thus, any term of type τ_o is preserved as τ_o , or as τ_n. We prove preservation by cases over the rewriting rules, R_1 and R_2 in Φ. For R_1, when a term has a type other than τ_o, Φ does not rewrite it. The preservation holds because it held in P. When a term has type τ_o, then it may be rewritten to τ_o (preserved), or τ_n. For τ_o case, the preservation trivially holds. For τ_n case, the preservation holds, since the expression and its evaluation have the same retypeabiliy, and so, in the evaluation the type is τ_n. The proof of preservation for R_2 is similar.

Φ modifies only types in P, so the evaluation of P' remains identical under Φ, modulo the user-defined operators. Because of the assumption that P progresses and the user-provided operators are correct, P' also progresses. $\square$

We continue using the `+` operator in Fig. 1c as an example to illustrate Φ. For unretypeable terms, where the cast $\dashv\!\exists$ is needed, we denote the new type constructed by $\dashv\!\exists$ with $\tau_{n\dashv\exists}$ for clearer presentation, which is essentially $\dashv\!\exists\ (\tau_o)$.

The type context on which Φ works is Γ, inferred by using $TA(\uplus, \cap, \dashv\!\exists, \omega)$. This contains the definitions for the user defined operators, as well as for the input program. Line `4` in Fig. 2 shows the user provided operator `+`: $op_1 op_2.body+ : \tau_n \rightarrow \tau_n \rightarrow \tau_n$. Line `7` defines the cast operator, used by $\dashv\!\exists$.

Further, we apply Φ on Γ, and obtain the rewritten type context Γ': $\Phi(\Gamma) = \Gamma' = \{\{op_1 : \tau_o \uplus \tau_n, op_1 : \tau_n\}, \{op_2 : \tau_o \uplus \tau_{n\dashv\exists}, op_2 : \tau_n\}, \{body+ : \tau_o \uplus \tau_n, body+ : \tau_n\}\}$. On Γ', we deduce the new type of every operator from the initial program. Using type inference in $TA(\uplus, \cap, \dashv\!\exists, \omega)$, we obtain that `+` has the type $\tau_n \rightarrow ((\tau_o \uplus \tau_{n\dashv\exists}) \cap (\tau_o \uplus \tau_n)) \rightarrow \tau_n$ in the retyped program. This type is inferred only from the type context in the input program, and we call it *general* type. RETYPE regards any type assignments that satisfy this type to be valid.

3.4 RETYPE's Type Checking

Φ constructs Γ', the rewritten type context of P, and τ_n. For type checking the retyped P, and deducing the actual types of the operators in the retyped program we check if the definitions in τ_n are consistent with Γ' .

We use constraint solving for the type check. The tc operator at line 4 (Algorithm 1) calls the constraint solver, for the type checking. The type assignments in Γ' are constraints on the actual types of terms in Γ'. For each term t, we check if the user provided operators in τ_n are a solution for all the constraints generated on t. If this is the case, we further have to guarantee that this is the unique solution. Otherwise, RETYPE cannot decide which solution is desired by the user. We do so, by intersecting the constraints and the complements of the returned solutions. We detail this in Sec. 4.

Constraint solving is exponential in some cases. However, the constraints generated by RETYPE are in conjunctive normal form by construction. The constraints before applying Φ are just intersections of the type assignments, in Γ, for each of the terms involving τ_o, so are in conjunctive normal form. Φ does two things: 1) R_1 introduce disjunction in the constraints; however these disjunctions are in parenthesis, and thus with a higher priority than the conjunctions. Each atomic type in the initial constraints generated by Γ might be replaced by disjunction with higher priority. Thus, the conjunctive normal form is preserved, since the constraints remain conjunctions of clauses, where a clause is a disjunction of literals. 2) R_2 simply replaces an atomic type, by using casts, with another one. This preserves the conjunctive normal form.

The generated constraints are guaranteed to be in conjunctive normal form, by construction, as shown above. The number of generated constraints is linear in the number of call sites, and variables. Each of those contain a type variable X, with two potential values: X, or $\neg X$. We map X to τ_n, and $\neg X$ to τ_o; thus, τ_o, and τ_n are literals in the constraints. Therefore, a constraint in RETYPE is a 2-SAT formula with a number of clauses on one type variable, involving two literals in each clause: τ_o, and τ_n. 2-SAT problems, are known to be solvable in polynomial time [26]. An example of constraint generated by RETYPE is: $\mathit{typeof}(x) = \tau_n \wedge \mathit{typeof}(x) = \tau_o$; this specifies that we need two definitions for the term x: one to be of the new type, and one to be of the old type.

If the constraints are satisfiable, and thus the program type checks, we proceed at rewriting annotations, and function calls if case, in P. Φ augments Γ' with all the rewritten types. Γ' holds the location information for every type assignment, denoted as l which is actually the line number in the input program. Thus, Γ' is a three-tuple: $\langle x : \tau, x' : \tau', l\rangle$, which indicates that the term x of the type τ at the location l is replaced with the term x' of type τ'. P becomes: $P[x : \tau, l \rightarrow x' : \tau', l], \forall (x : \tau, x' : \tau', l) \in \Gamma'$.

RETYPE treats variable definitions and function definitions differently. Specifically, it changes the annotations in variable definitions, but injects the operator definitions in τ_n at the end of the file, since the old function definition might still be used by unretyped parts of the input program. RETYPE adds new function definitions to the end of the source files, so that l remains identical between the input program and the retyped one. If m (the mapping between initial and new operator names) is missing, and thus with its identify default value, then Φ does not replace function calls, since them will be re-inferred by the base language's type system. Else Φ replaces the function calls, according with the user provided mappings in m . The last is always the case for languages that do not support operator overloadings, where m is required.

3.5 Semantics-Preservation under RETYPE

Retyping injects new behaviour into the retyped program by the means of the new operator definitions in τ_n. The relationship between the program's original and retyped behaviour depends on these definitions, whose semantics RETYPE does not constrain. When a user retypes an operator to one that crashes the program, for instance, RETYPE inevitably fails to preserve the original behaviour. However, when the users constrain the new behaviour of the operators in τ_n, we can prove semantics-preservation properties. First, we consider the case where τ_n adds operator definitions that introduce and operate only on fresh state, then we consider the case where the new operators change the program's original behaviour, but maintain a user-defined predicate over the original program's traces and those of its retyped incarnation.

For the first case, we use the Liskov substitution principle (LSP) [28]. Unlike nominal and structural subtyping, behavioural subtyping is defined on semantics instead of syntax and guarantees semantic interoperability of types in a hierarchy. Informally, LSP states that if S is a behavioural subtype of T, then objects of type T in a program may be replaced with objects of type S without affecting any provable property over objects of type T. Under LSP, retyping a variable from S to T preserves behaviour.

Let $\Sigma = X_0 \times X_1 \times \cdots \times X_k$ be the state space of the program P and $\Sigma' = \Sigma \times X_{k+1} \times \cdots X_j$ be the state space of a retyped variant of P. For $\vec{\sigma} \in \Sigma$ and $\vec{\sigma}' \in \Sigma'$, $=_\Sigma$ is program state equality restricted to Σ: $\vec{\sigma} =_\Sigma \vec{\sigma}' \equiv \bigwedge_{\sigma_i \in \vec{\sigma}} \sigma_i = \sigma'_i$. Let $\mathit{trace}(f(i)) = \sigma_0, \sigma_1, \cdots$ be the states generated by f's execution on i. Let ψ be a predicate over traces that filters out states internal to retyped operators, producing the traces $t = \sigma_0, \cdots, \sigma_k$ and $t' = \sigma'_0, \cdots, \sigma'_j$, then returns $i = j \wedge \bigwedge_{\sigma_i \in t} \sigma_i =_\Sigma \sigma'_i$.

Theorem 4 (Semantics-Preservation under the Liskov Substitution Principle)**.** *If τ_n is a subtype of τ_o under the Liskov substitution principle, then $\forall i \in \mathbf{dom}(P)$, $\psi(\mathit{trace}(P(i))$, $\mathit{trace}(\text{RETYPE}(P, \tau_o, \tau_n, m)(i)))$.*

Proof. LSP implies that any new behaviour τ_n defines operates on fresh data, not Σ. Noninterference (NI) partitions a program's inputs and outputs into low or high [19]. A program exhibits NI when changes in its high inputs do not change its low outputs. To apply NI to Theorem 4, we map the program's old state to low variables, and the new, "fresh" state, introduced by τ_n to high variables. We then prove noninterference by induction over applications of the new operators introduced by τ_n. In applying induction, we follow Focardi *et al.* in establishing step-wise NI, a stronger property that trivially implies NI [18].

In the base case, we show that all transitions from initial states in the retyped program are noninterferencing with respect to "fresh" state. We do so by showing that high states cannot flow into low ones across any operator in τ_n. This must hold under LSP, because, if not, failure for NI to hold is a provable property that would distinguish the retyped program's state space from the original program's state space when objects of type S replace objects of type T. Next, we assume that the noninterference holds over k applications of the new operators, and prove, again under LSP, that it holds for any $k+1$ application. Thus, no high or "fresh" state affects any low state: all evaluations outside of the retyped operators are low, or, in our case, original state equivalent. In particular, the evaluation of control point expressions produce low state, so control flow outside of retyped operators is unchanged. □

The LSP assumption handles naturally arising retype use cases, like taint analysis, refactoring, replacing an array with a list, logging, or instability detection in floating-point programs [2]. It is, however, too strong in many practical cases. Sometimes, we wish to retype a program to change its behaviour. For example, a developer may change `float` to `double` to increase a computation's accuracy. Under this retyping, the retyped program may take different execution paths on the same input, as when the original program exited upon overflowing a variable.

To handle these cases, we relax Theorem 4 in two ways. First, we allow the user to specify ψ to capture those semantics of the original program that τ_n preserves. Second, we only require ψ to hold over a non-empty subset $\hat{I} \subseteq \mathbf{dom}(P)$. When retyping `float` to `double`, a developer could define ψ to tolerate a bounded error, thus implicitly handling rounding errors (INEXACT exceptions) up to its bound. Consider Ariadne [6], which retypes a numerical program's floating-point variables to arbitrary precision rationals, reifies floating-point expressions to program points, then asks whether these program points are reachable. To use RETYPE, Ariadne would define ψ to match control expression evaluations and to ignore other program states except the final states of the two traces, which must be the same floating-point exception.

Theorem 5 (Semantics-Preservation modulo ψ)**.** If τ_n is a subtype of τ_o modulo ψ, then $\forall t \in \hat{I}$, $\psi(trace(P(i)), trace(retype(P, \tau_o, \tau_n, m)(i)))$.

Proof Idea. We use induction over operator applications, using case analysis to show that ψ holds over the τ_n's new operators at the base case and the inductive step. □

4. Implementation

We implement RETYPE based on ROSE [37], a compiler infrastructure on which developers can perform customised source-to-source transformation, analysis, and optimisation.

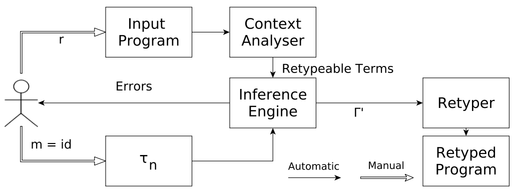


Figure 4: Retype Implementation Architecture

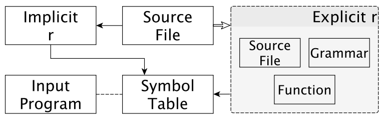


Figure 5: Retype Context Analyser

RETYPE operates on the input program's Abstract Syntax Tree, generated by ROSE. At a high level, RETYPE consists of three components (Fig. 4): first, the context analyser (Fig. 5) implements r, for annotating retypeable declarations or statements in the input program P; next, the inference engine (Fig. 6) takes the annotated P, infers the type context of the retypeable terms in P, applies Φ for rewriting the type context, and type checks the rewritings by using constraint solving; finally, the retyper (Fig. 7) rewrites the type annotations and function calls in the source code of the input program.

Context Analyser The primary task of the context analyser (Fig. 5), is to identify the retypeable terms (variables, functions and operators). It fulfils its task by implementing the retypeable function r, which further splits into implicit and explicit, as defined in Sec. 3.1. The context analyser identifies implicitly unretypeable terms by checking whether P contains a term's definitions. RETYPE permits developers to explicitly specify unretypeable terms in three ways: annotation, function, or file specifications.

The annotations specifications delimit the region to retype, and conceptually are not in the alphabet of programming language. In practice, we realise them using a unique sequence of characters, whose exact definition is configurable. Here we use `[*` and `*]` to denote these delimiters. RETYPE supports annotations at three granularities. First, if no annotations are found, RETYPE replaces the type for the entire program by default. Second, RETYPE supports statement granularity, such as: `[* w = a + foo(); *]`. Third, it supports inline annotation at variable granularity, such as: `int [* a *],b,w,x,[* y *],z;` .

The function specifications define retypeability in terms of functions. The user is able to limit the depth of the retyping: for a function name we retype just the current function; for a depth `k` we retype all the functions up to `k` depth in the call graph of the program, rooted in the current function; depth `*` means an infinite depth. For example, '`foo 1`' indicates `foo` and any directly called function to be retyped. File specifications annotates the entire code in a file as retypeable.

For all r's specifications, the user can also specify their complement (*i.e.* what not to retype, rather than what to retype). After identifying all the retypeable terms, the context

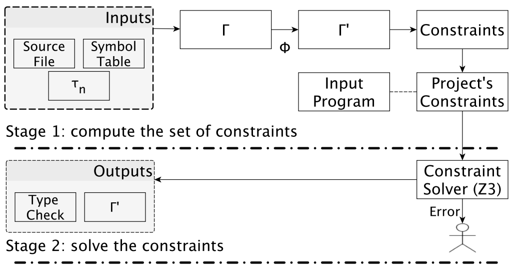


Figure 6: Retype Inference Engine

analyser annotates the program constructs that contain such terms, in a table, ‘ST’ in Fig. 4.

Inference Engine As RETYPE’s main component, the inference engine realises the type assignment, the term rewriting system Φ, and the constraint solving.

Unification [22] and constraint solving [35] are two ways to implement type inference and type checking. Unification generates fresh type variables for all the types in the program, and then finds substitutions for that bind each type variable to a type expression. The set of substitutions is called the unifier. The unification algorithm aims at identifying the most general unifier. Initially each type expression is unbound. Constraint solving takes as input a set of constraints, generated by each usage of a term in the program, and then calls a constraint solver to produce solutions for them.

RETYPE uses constraint solving, because it allows us to precisely report errors. When calling the solver, we know all the constraints on types. Thus, RETYPE can accurately report the location and reason for an error when a constraint is unsatisfiable. Detailed error messages improve RETYPE’s usability. We did not use unification because its evaluation is non-deterministic so the error depends on the unification order.

As Fig. 6 shows, the inference engine comprehends two stages. The first stage is syntax-directed. The inference engine traverses P, annotated by the context analyser, introduces type variables for each term, and builds a constraint for every use of a term involving τ_o. Then the inference engine collects all the constraints, and stores each program construct involved in constraints, together with the location.

The second stage first eliminates identical constraints on the same term by using a constraint solver, specifically, Z3 [38]. Then the inference engine applies the two rules of Φ on the constraints, introducing $\uplus$, and $\dashv$, as detailed in Sec. 3.3. Next, it calls Z3 to solve the constraints. The solution represents the new type of the corresponding operator. In the event of the solver returning an unknown, we notify the user but fail to provide any detailed error messages. To date, we have not observed any unknowns. If there is no solution, the solver returns the unsatisfiable constraint. RETYPE checks the constraint’s information stored in the previous stage, and provides the user with its location and the reason why it fails. Further it confirms that the constraint has only one solution; otherwise RETYPE cannot decide which solution the user desires.

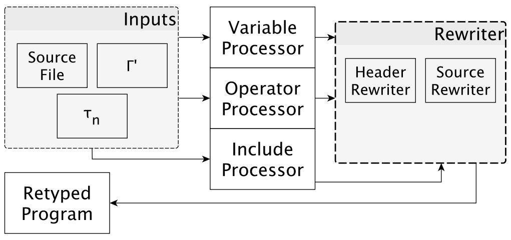


Figure 7: Retype Program Transformation Architecture

It confirms this by intersecting the constraint and the complement of the returned solution. If the solver produces another solution, we report an error. Finally, the inference engine maps the user provided operators to the program constructs corresponding to the constraints involving them, and feed the map to the *Retyper*. The solution to the constraints should replace the corresponding program constructs. ‘`int` $\rightarrow$ `TaintInt`, at line 2, file main.c’ is a typical entry in the mapping.

Retyper Based on the mapping that the inference engine generates, the retyper (Fig. 7) simply traverses P, changes the type signatures, and replaces the initial operator calls with calls to the ones provided by the user. The retyper adds the new function definitions to the end of the source files to keep the code locations identical in the retyped program. Every time the retyper applies a cast on a variable that follows a reference operator, it constructs a fresh variable of the new type to store the variable’s value in the line before, and casts it back afterwards. It does so, for allowing external functions to update the value of a reference. This is not possible with a direct cast, in the actual argument of an external function (Sec. 2).

5. Evaluation

Our evaluation first establishes that RETYPE works, that it changes a program’s type annotations without otherwise disrupting that program. Then we demonstrate RETYPE’s utility, by using it to automate the retyping tasks in taint analysis and energy reduction, and comparing it with the retyping support in Eclipse and IntelliJ IDEA. Finally, we show RETYPE scales to `Pidgin`, a program whose LOC is 363K.

5.1 Sanity Check

If RETYPE is correct, a roundtrip application of RETYPE, $\text{RETYPE}(\text{RETYPE}(P, \tau_o, \tau_n), \tau_n, \tau_o)$ (that is, $f \circ f^{-1}$, for $f = \text{RETYPE}$) should return a program that is semantically equivalent to the input program.

A problem naturally arises: if some terms already use τ_n in a retypeable region of the original program, the second retyping process might incorrectly change them which should remain the same. Similar to α-renaming [45], RETYPE solves this problem by always introducing fresh type names for t_n, such as using `typedef` in C, to avoid unintended rewriting.

In the sanity check, we use testing to under-approximate the programs’ semantics. we select five open source programs that have test suites with relatively high statement coverage, 77.4% on average, two written in C and three in Java. For

Project	Statement Coverage	Retyping Successful?	Sanity Check?
Commons Email 1.4	69%	Yes	Yes
Commons Validator 1.4.4	83%	Yes	Yes
fwknop 2.6.6	90%	Yes	Yes
Joda Time 2.1	90%	Yes	Yes
OpenSSH 6.6	55%	Yes	Yes

Table 1: The sanity check results. *Retyping Successful?* shows if RETYPE successfully changed the types twice; *Sanity Check?* reports if the retyped programs passed the tests.

each program, we build two sets of types and randomly select a pair (τ_o, τ_n). We apply RETYPE without defining r that specifies retypeable terms, first changing τ_o to τ_n and then τ_n to τ_o. We observe that all five transformed programs pass all the tests. Table 1 reports the results of this sanity check.

5.2 The Utility of RETYPE

Developers care about taint analysis and energy consumptions. Can RETYPE help semi-automate these tasks? To demonstrate RETYPE's utility, we replicate the results of programs from these domains. For taint analysis, we add two lines to the program in Fig. 3 to taint variables `a` and `b`: `1 a.tainted = true;`, and `2 b.tainted = true;` Fig. 2 presents a complete definition of `+` to illustrate how taint propagates. Finally, we output the fields `value` and `tainted` of variables `w`, `x`, `y` and `z`. As expected, all these four variables are tainted. Through computations, the taint propagates correctly from the source to the other four variables. Meanwhile, their values remain the same as in the original program: `w = {22, tainted}; x = {22, tainted}; y = {5, tainted}; z = {28, tainted}`.

SEEDS [30] is a general framework that aims to reduce a program's energy usage. The authors instantiate SEEDS by automatically switching among the subclasses of Java's Collections library to select the most energy efficient implementation; the switching is accomplished by rewriting bytecode. The experimental results suggest that energy consumption can be reduced by up to 17%.

To partially replicate SEEDS, we identify all locations where a retyping is possible and repeatedly apply RETYPE to search for a more energy efficient implementation from the Java Collections Framework. In the SEEDS work, the authors used to automatically identify τ_o and τ_n and did not record their selection. To reproduce SEEDS, we need to manually identify this pair in a large space of combinations. Though RETYPE automates the type replacement, the overall task remains manual. Therefore, we select only a benchmark used in their preliminary study and an experiment subject, Commons Cli. The benchmark [27] was originally composed to compare the performance of different Collection implementations. We have modified it so that only `LinkedList` is used, and then we apply RETYPE to generate new versions using other implementations, such as `ArrayList` and `Vector`. Commons Cli [41] is an open source project that parses the command line options.

SEEDS' authors showed that the greatest improvement of energy consumption for Commons Cli results from a single

Program	Reduction	$\tau_o \rightarrow \tau_n$
Collection Benchmark	83.6%[2]	`LinkedList` $\rightarrow$ `ArrayList`
Commons Cli 2.9.1	7.2%	`ArrayList` $\rightarrow$ `Vector`

Table 2: RETYPE's application for optimising energy consumption. *Energy Reduction* shows how much energy consumption is reduced by using the most energy efficient implementation; $\tau_o \rightarrow \tau_n$ describes the actual retyping.

implementation replacement, so we do not consider more than one retyping. Due to the lack of hardware, we use Jalen [34], a Java agent that monitors an application's energy consumption, for the measurement. Table 2 presents our result. In SEEDS, selecting an appropriate implementation reduced the energy consumption by 2%; while our result suggests up to 30% of the energy can be saved. We believe that replacing hardware with software measurement accounts for this discrepancy.

5.3 Comparison against Eclipse and IntelliJ IDEA

Having established the baseline of its utility, we compare RETYPE with the retyping functionality of two popular IDEs, Eclipse and IntelliJ IDEA. Eclipse implements *change method signature*, allowing one to change the types of a method's parameters or return value. For retyping to succeed, Eclipse checks if the method's new type signature conforms with its call sites and if the new type can be used in the method body. If the new types are explicit subtypes of the old ones, these checks pass and Eclipse replaces only the type annotation in the method's signature. Otherwise Eclipse reports an error. Because Eclipse's retyping is so limited, Table 3 presents the comparison of RETYPE and only Type Migration from IntelliJ IDEA. This comparison, restricted to Java of which IntelliJ is an IDE, is under six dimensions.

IntelliJ allows the user to **specify retypeable terms** at the granularities of variable, file, and project; RETYPE provides two more, line and function. Changing a certain type, for all the variables in the specified regions, is possible only in RETYPE. Unlike RETYPE, IntelliJ does not handle **program constructs**, such as literal constants and constructors. For example, when we gave IntelliJ code containing constants whose types needed to be changed, IntelliJ simply showed the retype operation as impossible. IntelliJ is an IDE for Java, and thus it supports only the OO paradigm; RETYPE is not restricted to the OO paradigm and the **language support** of its current implementation rests on ROSE. IntelliJ requires τ_n to be inherited from τ_o using the keyword `extend`; RETYPE, with an optional mapping m, does not constrain as strictly **the relation of τ_o and τ_n**, being able to retype cases, like subtyping without `extend` or a pair of types with different method or filed names. When a retyping is unsuccessful, IntelliJ simply exits without useful **error information**, and it cannot identify errors raised when one retypes the arguments or returns

[2] When using HashSet, the benchmark runs too fast for Jalen to produce any useful measurement.

	Retypeable Specification	IJ	R
1	Granularity of Variable	Yes	Yes
2	Granularity of Type	No	Yes
3	Granularity of Project	Yes	Yes
4	Granularity of File(s)	Yes	Yes
5	Granularity of Function(s)	No	Yes
6	Granularity of Line(s)	No	Yes

	Handled Program Constructs	IJ	R
1	Literal Constants	No	Yes
2	Constructors	No	Yes

	Supported PLs	IJ	R
1	OO: Java	Yes	Yes
2	Procedural: C	No	Yes
3	Multi-Paradigm: C++	No	Yes

	$\tau_o \to \tau_n$ relation	IJ	R
1	Explicit Subtype with Inheritance	Yes	Yes
2	Subtype without Inheritance	No	Yes
3	$name(\tau_o) \neq name(\tau_n)$	Yes	Yes
4	Different Method Names	No	Yes
5	Different Field Names	No	Yes

	Error Handling	IJ	R
1	Error Location	No	Yes
2	Error Reason	No	Yes
3	Call Site Sensitive	No	Yes

	Unretypeable Calls	IJ	R
1	Cast Incoming Values	No	Yes
2	Cast Outcoming Values	No	Yes

Table 3: The results of the comparison between RETYPE (R) and the type migration feature in a popular IDE: IntelliJ IDEA (IJ).

Category	All Program		One Variable	
	Test Cases	Count Fields	Test Cases	Count Fields
Prim $\to$ Prim	100%	225	100%	7
Prim $\to$ ADT	100%	439	100%	16
ADT $\to$ Prim	100%	439	100%	16
ADT $\to$ ADT	100%	173	100%	4

Table 4: Scalability and effectiveness of RETYPE. *Category* shows the categories of τ_o, and τ_n. This can be Abstract Data Types (ADT) or Primitive Data Types (Prim). For example retyping from primitive to ADT is written as: 'Prim $\to$ ADT'.

of functions at a non-retypeable call site; RETYPE explains what causes the error and its exact location, and RETYPE is call site sensitive. IntelliJ does not permit retyping for terms that **flow into or out of retypeable regions**; RETYPE uses casts to handle these.

5.4 RETYPE's Scalability

A real-world program may contain millions lines of code. A retyping tool that does not scale is impractical, even if it is correct and useful. To show RETYPE's scalability, we apply it to `Pidgin`, an open-source online chatting client which contains 363K lines of code. In general, a type replacement is one of the four kinds: primitive to primitive, primitive to ADT, ADT to primitive and ADT to ADT. We identify five concrete type replacements for each kind that obey the Liskov substitution principle. RETYPE then performs the retyping one by one, and runs `Pidgin`'s test cases to see to what extent the programming behaviour is affected. We find that RETYPE has made up to 439 modifications. As expected, RETYPE passes all the test, because all of the new types we identified only introduce new behaviours, such as changing `int` to `TaintInt`. Theorem 4 guarantees semantics-preservation under LSP. Our results empirically validate the theorem. Also, testing is an under-approximation of the program behaviour. Table 4 presents the detailed outcome of the experiment.

6. Related Work

Programmers' desire to change a type has captured industry's attention, reflected by the fact that two popular IDEs have built-in yet limited support for this feature. IntelliJ IDEA [23] implements type migration, capable of changing the type of a variable or a method by performing inter-procedural data flow analysis [24, 36]. Eclipse's [15] retyping functionality is restricted to changing a method's signature [17].

Static and dynamic type systems have their own strengths and weaknesses. Gradual typing [40] aims to combine them, allowing a developer to partially annotate their programs. Specifically, it statically type checks annotated program regions and dynamically type checks the remainder. For a language that supports partial annotation, like TypeScript, over the inputs on which a program runs without type errors, gradual typing is a restricted form of retyping from the missing or empty type to the type annotated by the developer.

Balaban *et al.* proposed Class Migration [1], later described as ReplaceClass [42, 43], a refactoring technique that uses type constraints [39] to change an existing class to a semantically equivalent one. A constraint variable represents the type to which an expression evaluates. A type constraint captures subtype relations between constraint variables. Subtyping does not order many types, like primitives or `Vector` and `ArrayList` in Java. To extend type constraints to handle these types, the authors add an operator $\not\leq$ which expresses the relations between constraint variables that are not ordered in a subtype lattice. Further, they introduce an operator $\in$ and implication constraints for considering two distinct type constraints generated by the original expression and its rewrite. For example, $\alpha_1 \in \{$`Vector`, `ArrayList`$\}$ denotes α_1 is either a `Vector` or a `ArrayList`, and $\alpha_1 =$ `ArrayList` $\Rightarrow \alpha_2 \leq \alpha_3$ means if α_1 is bound to `ArrayList`, then α_2 must be a subtype of α_3. Given the original and new class, ReplaceClass solves a system of type constraints generated from the original program. If there exists a type assignment, other than the original one which always is a valid solution, maximising the number of replacements, ReplaceClass applies it.

ReplaceClass aims to be semantics-preserving; RETYPE does not. Thus, while RETYPE can, like ReplaceClass, migrate a code base to a new version of a library and be semantics-preserving, RETYPE can and often does introduce new behaviour in order to optimise, as when changing the width of variables, or analyse the original program, such as to support taint analysis or transform the program to produce symbolic constraints amenable to solution by SMT solvers. ReplaceClass assumes an Object-Oriented (OO) language that supports subtyping, while RETYPE implements subtyping itself, allowing it to operate on non-OO languages, like C. ReplaceClass currently does not handle operator overloading and it would be adapted to rewrite the type constraints to introduce fresh names; RETYPE naturally and directly supports overloadings via $\cap$-types. RETYPE automatically injects casts to handle external functions and constants. If ReplaceClass added support for casts, their solution space would explode, since casting to the original type is always possible. Due to backtracking, the worst-case time complexity of ReplaceClass is exponential in the number of call sites, while that of RETYPE is polynomial. However, ReplaceClass considers preserving synchronisation using escape analysis [8]; RETYPE does not.

7. Conclusion and Future Work

We have defined retyping, a process for changing program behaviour under the control of a type system. To retype a program is to replace one type in it with another, replacing operator applications that involve the new type. We have presented a β-equivalent type assignment system and an accompanying decidable, sound term rewriting system that formalises retyping. We have presented and validated RETYPE, a tool for retyping programs. Upon acceptance of this work, we will make RETYPE available to the community at `www.utopia.com`; it is our hope that RETYPE will facilitate research by eliminating the need to reinvent the wheel by implementing bespoke, unsound retypings. To enhance RETYPE's shareability and reduce the effort in reproducing the development environment, we utilise Docker [14], a technique that virtualises operating systems by providing an additional layer of abstraction, to encapsulate RETYPE. We plan to supplement RETYPE with a library of types and operators, like tainted versions of common types, that would be useful retyping targets.

References

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 265–279, New York, NY, USA, 2005. ACM.

[2] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 817–832, New York, NY, USA, 2013. ACM.

[3] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In *Theoretical Aspects of Computer Software*, pages 651–674. Springer, 1991.

[4] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[5] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015. To appear.

[6] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 549–560, New York, NY, USA, 2013. ACM.

[7] L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[8] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[9] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940.

[10] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[11] H. Comon, G. Godoy, and R. Nieuwenhuis. The confluence of ground term rewrite systems is decidable in polynomial time. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 298–307. IEEE, 2001.

[12] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

[13] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *CAAP'92*, pages 102–123. Springer, 1992.

[14] D. Developers. Docker. `https://www.docker.com`, 2015. Accessed: 2015-07-08.

[15] E. Developers. Eclipse. `http://www.eclipse.org`, 2015. Accessed: 2015-07-08.

[16] P. Developers. Pidgin. `https://pidgin.im`, 2015. Accessed: 2015-07-08.

[17] Eclipse. Refactoring support. `http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm`, 2015. Accessed: 2015-07-08.

[18] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Foundations of Software Science and Computational Structures*, pages 299–315. Springer, 2005.

[19] J. A. Goguen and J. Meseguer. *Security policies and security models*. IEEE, 1982.

[20] J. R. Hindley. The simple semantics for coppo-dezani-sallé types. In *International symposium on programming*, pages 212–226. Springer, 1982.

[21] J. R. Hindley. *Basic simple type theory*. Number 42 in Type theory. Cambridge University Press, 1997.

[22] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[23] IntelliJ. Intellij idea. `https://www.jetbrains.com/idea/`, 2015. Accessed: 2015-07-08.

[24] M. Khalusova. Type migration refactoring. `http://blog.jetbrains.com/idea/2008/06/type-migration-refactoring`, 2015. Accessed: 2015-07-08.

[25] J. W. Klop and J. Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[26] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13(1-2):15–20, 1967.

[27] L. Lewis. Java collections performance. `http://lewisleo.blogspot.jp/2012/08/java-collections-performance.html`, 2015. Accessed: 2015-07-08.

[28] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[29] R. Loader. Higher order beta matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.

[30] I. Manotas, L. Pollock, and J. Clause. Seeds: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, pages 503–514. ACM, 2014.

[31] A. Marginean, E. T. Barr, M. Harman, and Y. Jia. Automated transplantation of call graph and layout features into kate. In *Search-Based Software Engineering*, pages 262–268. Springer, 2015.

[32] N. Nemo and J. Doe. The anonymous technical report. Technical report, Utopia, 3546.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *CMU*, 2005.

[34] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 160–169. IEEE, 2012.

[35] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and practice of object systems*, 5(LAMP-ARTICLE-1999-001):35, 1999.

[36] E. Pragt. Intellij 8: Type migration. http://blog.xebia.com/2008/11/07/intellij-8-type-migration, 2015. Accessed: 2015-07-08.

[37] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[38] M. Research. Z3. https://z3.codeplex.com, 2015. Accessed: 2015-07-08.

[39] M. I. Schwartzbach and J. Palsberg. *Object-oriented type systems*. Wiley., 1994.

[40] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[41] A. Team. Common cli. https://commons.apache.org/proper/commons-cli/, 2015. Accessed: 2015-07-08.

[42] F. Tip. Refactoring using type constraints. In *Static Analysis*, pages 1–17. Springer, 2007.

[43] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(3):9, 2011.

[44] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *ACM SIGPLAN Notices*, volume 38, pages 13–26. ACM, 2003.

[45] F. Turbak, D. Gifford, and M. A. Sheldon. *Design concepts in programming languages*. MIT press, 2008.

Bibliography

[1] S. Aaronson. Guest column: Np-complete problems and physical reality. *ACM Sigact News*, 36(1):30–52, 2005.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[3] M. Alan. Intelligent machinery. *Machine intelligence*, 5:3–23, 1992.

[4] M. Allamanis and M. Brockschmidt. Smartpaste: Learning to adapt source code. *arXiv preprint arXiv:1705.07867*, 2017.

[5] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[6] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin. Deploying search based software engineering with Sapienz at Facebook (keynote paper). In 10^{th} *International Symposium on Search Based Software Engineering (SSBSE 2018)*, pages 3–45, Montpellier, France, September 8th-10th 2018. Springer LNCS 11036.

[7] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12. IEEE, 2011.

[8] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, 2013.

[9] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[11] K. V. Ambrose, A. M. Koppenhöfer, and F. C. Belanger. Horizontal gene transfer of a bacterial insect toxin gene into the epichloë fungal symbionts of grasses. *Scientific reports*, 4, 2014.

[12] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[13] Androidmonkey. http://developer.android.com/guide/developing/tools/monkey.html, 2016. Accessed: 2020-01-24.

[14] P. J. Angeline and K. E. Kinnear. Parallel genetic programming. *Springer*, 1996.

[15] ansible-xml Developers. ansible-xml. https://github.com/cmprescott/ansible-xml, 2016. Accessed: 2016-01-12.

[16] AppLove Community. AppLove. https://github.com/snowpunch/AppLove, 2016. Accessed: 2016-01-12.

[17] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *IEEE Congress on Evolutionary Computation (CEC'08)*, pages 162–168, Hongkong, China, 1-6 June 2008. IEEE Computer Society.

[18] A. Arcuri and X. Yao. Co-evolutionary automatic programming for software development. *Information Sciences*, 259:412–432, 2014.

[19] Argparse. https://github.com/python/cpython/blob/master/Lib/argparse.py, 2016. Accessed: 2020-01-24.

[20] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[21] D. Ashlock, C. McGuinness, and W. Ashlock. Representation in evolutionary computation. In *Proceedings of the 2012 World Congress Conference on Advances in Computational Intelligence*, WCCI'12, pages 77–97, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[23] T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[24] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. CRC Press, 1997.

[25] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, Dec. 2001.

[26] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 292–303. IEEE, 1999.

[27] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 98–107. IEEE, 2000.

[28] H. Barendregt. Lambda calculi with types, handbook of logic in computer science (vol. 2): background: computational structures, 1993.

[29] M. Barlow. What antimicrobial resistance has taught us about horizontal gene transfer. In *Horizontal Gene Transfer*, pages 397–411. Springer, 2009.

[30] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.

[31] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[32] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93)*, pages 509–518, Los Alamitos, California, USA, 1993. IEEE Computer Society Press.

[33] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[34] T. V. Belle and D. H. Ackley. Code factoring and the evolution of evolvability. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1383–1390. Morgan Kaufmann Publishers Inc., 2002.

[35] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[36] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti. Portable, high-performance containers for hpc. *arXiv preprint arXiv:1704.03383*, 2017.

[37] A. Beszédes and T. Gyimóthy. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance*, pages 12–20, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[38] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In 4th *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 43–52, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[39] D. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[40] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In 22nd *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, Hong Kong, China, November 2014.

[41] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo. Orbs: Language-independent program slicing - website. http://crest.cs.ucl.ac.uk/resources/orbs. Accessed: 2015-06-13.

[42] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo. Observation-based slicing. Technical Report RN/13/13, Computer Sciences Department, University College London (UCL), UK, June 20th 2013.

[43] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[44] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *The second annual object-oriented numerics conference (OON-SKI*. Citeseer, 1994.

[45] B. Bolick, G. Gopalakrishnan, C. Jacobsen, and T. Tuttle. Toward revamping discrete structures: Concurrency through functional/constraint programming integration. *Discrete Structures*, 2014.

[46] I. BoussaïD, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[47] P. J. Bowler. *Evolution: the history of an idea*. Univ of California Press, 1989.

[48] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 13–24. IEEE, 2011.

[49] M. Brameier and W. Banzhaf. Linear genetic programming. genetic and evolutionary computation, vol. xvi, 2007.

[50] K. A. Broughan, G. Keady, T. D. Robb, M. G. Richardson, and M. C. Dewar. Some symbolic computing links to the NAG numeric library. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 25(3):28–37, July 1991.

[51] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1327–1334. ACM, 2015.

[52] W. S. Bruce. The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*. Citeseer, 1997.

[53] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[54] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[55] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[56] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium*, pages 3–11, 2015.

[57] C. Calcagno, D. Distefano, and P. O'Hearn. Open-sourcing Facebook Infer: Identify bugs before you ship. code.facebook.com blog post, 11 June 2015.

[58] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011. Preliminary version appeared in POPL'09.

[59] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

[60] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance*, pages 424–433, Los Alamitos, California, USA, Sept. 1994. IEEE Computer Society Press.

[61] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. In 6^{th} *IEEE International Workshop on Program Comprehension*, pages 136–144, Los Alamitos, California, USA, June 1998. IEEE Computer Society Press.

[62] G. Canfora, A. De Lucia, and M. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[63] G. Canfora and M. Di Penta. New frontiers in reverse engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[64] C. K. Chang. Changing face of software engineering. *IEEE Software*, 11(1):4–5, 1994.

[65] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[66] C. Chauhan, R. Gupta, and K. Pathak. Survey of methods of solving tsp along with its implementation using dynamic programming approach. *International Journal of Computer Applications*, 52(4), 2012.

[67] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[68] A. Cimitile, A. R. Fasolino, and P. Marascea. Reuse reengineering and validation via concept assignment. In *International Conference on Software Maintenance (ICSE 1993)*, pages 216–225. IEEE Computer Society Press, Sept. 1993.

[69] clawpack Developers. clawpack. `https://github.com/clawpack/clawpack`, 2016. Accessed: 2016-01-12.

[70] R. Coelho, L. Almeida, G. Gousios, A. Van Deursen, and C. Treude. Exception handling bug hazards in android. *Empirical Software Engineering*, 22(3):1264–1304, 2017.

[71] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[72] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. NPEFix: Automatic runtime repair of null pointer exceptions in java. working paper or preprint, 2015.

[73] C. Darwin. *On the origin of species, 1859*. Routledge, 2004.

[74] C. Developers. Compress. `https://github.com/ning/compress`, 2016. Accessed: 2016-01-12.

[75] E. developers. Exchangerateapp. `https://github.com/denniss/ExchangeRateApp`, 2016. Accessed: 2016-01-12.

[76] F. Developers. Flap. `https://github.com/szaghi/FLAP`, 2016. Accessed: 2016-01-12.

[77] N. Developers. Newsreader. https://github.com/elliottetzkorn/NewsReader, 2016. Accessed: 2016-01-12.

[78] N.-C. developers. Nsdata-compression. https://github.com/leemorgan/NSData-Compression, 2016. Accessed: 2016-01-12.

[79] O. Developers. Owasp. https://github.com/OWASP/java-html-sanitizer, 2016. Accessed: 2016-01-12.

[80] P. Developers. Pydas. https://github.com/jwallen/PyDAS, 2016. Accessed: 2016-01-12.

[81] P. Developers. Pliny project. http://pliny.rice.edu, 2018.

[82] S. Developers. Slycot. https://github.com/avventi/Slycot, 2016. Accessed: 2016-01-12.

[83] S. C. Developers. Sip - calculator. https://github.com/tirupati17/sip-calculator-swift, 2016. Accessed: 2016-01-12.

[84] T. Developers. Tswechat. https://github.com/hilen/TSWeChat, 2016. Accessed: 2016-01-12.

[85] T. O. P. A. Developers. The oakland post app. https://github.com/aclissold/the-oakland-post, 2016. Accessed: 2016-01-12.

[86] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

[87] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software: Evolution and Process*, 18(2):83–107, 2006.

[88] Docker. Docker. https://www.docker.com. Accessed: 2015-06-17.

[89] Docker. Docker security and best practises. https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices. Accessed: 2015-06-17.

[90] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling salesman problem. *biosystems*, 43(2):73–81, 1997.

[91] B. J. Dorr, P. W. Jordan, and J. W. Benoit. A survey of current paradigms in machine translation. In *Advances in computers*, volume 49, pages 1–68. Elsevier, 1999.

[92] K. Dowd, C. Severance, and M. Loukides. *High performance computing*. O'Reilly & Associates, Inc., 1998.

[93] dropship Developers. dropship. https://github.com/driverdan/dropship, 2016. Accessed: 2016-01-12.

[94] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 118–128. ACM, 2007.

[95] G. M. Edelman. *Neural Darwinism: The theory of neuronal group selection.* Basic books, 1987.

[96] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3), 2003. Special issue on ICSM 2001.

[97] Equationsolver. https://docs.python.org/3/library/json.html, 2016. Accessed: 2020-01-24.

[98] Facebook. Facebook research post describing the move of majicke to facebook. https://facebook.com/academics/posts/1326609954057075, 2017. Accessed: 2020-01-24.

[99] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[100] Y. Feng, Y. Wang, R. Martins, A. A. Kaushik, I. Dillig, M. Ali, R. Reaz, M. Gouda, J. Huang, T. M. Smith, et al. Type-directed component-based synthesis using petri nets. Technical report, Technical Report TR-16-01, Department of Computer Science, UT-Austin, 2016.

[101] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[102] Firefox-iOS Developers. Firefox-iOS. https://github.com/mozilla-mobile/firefox-ios, 2016. Accessed: 2016-01-12.

[103] M. Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[104] fortranlib Developers. fortranlib. https://github.com/astrofrog/fortranlib, 2016. Accessed: 2016-01-12.

[105] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[106] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130. IEEE, 2012.

[107] fson Developers. fson. https://github.com/josephalevin/fson, 2016. Accessed: 2016-01-12.

[108] J. Y. Gil and I. Maman. Micro patterns in java code. *ACM SIGPLAN Notices*, 40(10):97–116, 2005.

[109] Github. Github. https://github.com, 2016. Accessed: 2016-01-12.

[110] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[111] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE 2012)*, pages 3–13, Zurich, Switzerland, 2012.

[112] C. Green. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier, 1981.

[113] S. Green, D. Cer, and C. D. Manning. Phrasal: A toolkit for new directions in statistical machine translation. In *In Proceddings of the Ninth Workshop on Statistical Machine Translation*, 2014.

[114] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46(1):317–330, Jan. 2011.

[115] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[116] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[117] S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[118] S. Haefliger, G. Von Krogh, and S. Spaeth. Code reuse in open source software. *Management science*, 54(1):180–193, 2008.

[119] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, Mar. 1995.

[120] M. Harman. The current state and future of search based software engineering (invited paper). In *29th International Conference on Software Engineering (ICSE 2007), Future of Software Engineering (FoSE)*, Minneapolis, USA, 2007.

[121] M. Harman. Software engineering meets evolutionary computation. *Computer*, 44(10):31–39, 2011.

[122] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[123] M. Harman, N. Gold, R. M. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 11 – 21, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[124] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[125] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing (keynote paper). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015.

[126] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[127] M. Harman, W. B. Langdon, and Y. Jia. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In 6^{th} *Symposium on Search Based Software Engineering (SSBSE 2014)*, Fortaleza, Brazil, August 2014. Springer LNCS.

[128] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[129] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering (keynote paper). In R. Oliveto and R. Robbes, editors, 20^{th} *Working Conference on Reverse Engineering (WCRE 2013)*, Koblenz, Germany, 14-17 October 2013. IEEE.

[130] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[131] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[132] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis (keynote paper). In 18th *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*, Madrid, Spain, September 23rd-24th 2018. To Appear.

[133] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In 20th *International Conference on Software Engineering (ICSE '98)*, pages 74–83. IEEE Computer Society Press, Apr. 1998.

[134] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *24th International Symposium on High-Performance Computer Architecture (HPCA 2018), February 24-28, Vienna, Austria*, 2018.

[135] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[136] C. A. R. Hoare. Null references: The billion dollar mistake. In *QCON conference*, London, England, 2009.

[137] K. L. Hoffman, M. Padberg, and G. Rinaldi. Traveling salesman problem. In *Encyclopedia of operations research and management science*, pages 1573–1578. Springer, 2013.

[138] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. *CSAIL Technical Reports*, 2009.

[139] J. Holland. Adaptation in natural and artificial systems: an introductory analysis with application to biology. *Control and artificial intelligence*, 1975.

[140] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[141] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, 2014.

[142] J. Jacobellis, N. Meng, and M. Kim. Lase: an example-based program transformation tool for locating and applying systematic edits. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1319–1322. IEEE Press, 2013.

[143] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

[144] jGnash Developers. jgnash. https://github.com/ccavanaugh/jgnash, 2015. Accessed: 2015-07-08.

[145] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[146] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 2008. IEEE Computer Society.

[147] Y. Jia and M. Harman. Higher order mutation testing. *Journal of Information and Software Technology*, 51(10):1379–1393, 2009.

[148] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

[149] Y. Jia, M. Harman, W. B. Langdon, and A. Marginean. Grow and serve: Growing Django citation services using SBSE. In *International Symposium on Search Based Software Engineering*, pages 269–275. Springer, 2015.

[150] L. Kallel and M. Schoenauer. Alternative random initialization in genetic algorithms. In *ICGA*, pages 268–275, 1997.

[151] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[152] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184. ACM, 2014.

[153] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[154] M. A. Kay, J. C. Glorioso, and L. Naldini. Viral vectors for gene therapy: the art of turning infectious agents into vehicles of therapeutics. *Nature medicine*, 7(1):33–40, 2001.

[155] B. Kazimipour, X. Li, and A. K. Qin. A review of population initialization techniques for evolutionary algorithms. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 2585–2592. IEEE, 2014.

[156] P. J. Keeling and J. D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 9(8):605–618, 2008.

[157] S. Kell. A survey of practical software adaptation techniques. *J. UCS*, 14(13):2110–2157, 2008.

[158] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In 35th *International Conference on Software Engineering (ICSE'13)*, pages 802–811, San Francisco, CA, USA, 2013. IEEE / ACM.

[159] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon. Facoy–a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[160] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

[161] P. Koehn, F. J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics, 2003.

[162] A. Kolmogoroff. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35(1):58–65, 1932.

[163] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In 27th *Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., Jan. 19–21 2000. ACM Press.

[164] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[165] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[166] R. Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[167] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[168] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[169] J. Krinke. Barrier slicing and chopping. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[170] J. Krinke. Is cloned code older than non-cloned code? In J. R. Cordy, K. Inoue, S. Jarzabek, and R. Koschke, editors, 5^{th} *ICSE International Workshop on Software Clones, IWSC 2011*, pages 28–33, Waikiki, Honolulu, HI, USA, 2011. ACM.

[171] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[172] Y. Kwon, W. Wang, Y. Zheng, X. Zhang, and D. Xu. Cpr: Cross platform binary code reuse via platform independent trace program. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 158–169. ACM, 2017.

[173] Y. Kwon, X. Zhang, and D. Xu. Pietrace: Platform independent executable trace. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 48–58. IEEE, 2013.

[174] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[175] W. B. Langdon. Genetic improvement of software for multiple objectives. In *International Symposium on Search Based Software Engineering*, pages 12–28. Springer, 2015.

[176] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.

[177] W. B. Langdon and M. Harman. Genetically improved CUDA C++ software. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[178] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, 19(1):118 – 135, February 2015.

[179] W. B. Langdon, B. Y. H. Lam, M. Modat, J. Petke, and M. Harman. Genetic improvement of gpu software. *Genetic Programming and Evolvable Machines*, 18(1):5–44, 2017.

[180] W. B. Langdon, M. Modat, J. Petke, and M. Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 Conference on Genetic*

and Evolutionary Computation, GECCO ’14, pages 951–958, New York, NY, USA, 2014. ACM.

[181] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[182] P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[183] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[184] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[185] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[186] O. A. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 212–221. ACM, 2014.

[187] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294–306, 2011.

[188] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes. Can the use of types and query expansion help improve large-scale code search? In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 41–50. IEEE, 2015.

[189] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[190] F. Lianubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[191] C. Liao, D. Quinlan, and T. Panas. Towards an abstraction-friendly programming model for high productivity and high performance computing. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.

[192] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-based openmp 3.0 research compiler supporting multiple runtime libraries. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 15–28. Springer, 2010.

[193] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*, pages 308–322. Springer, 2010.

[194] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas. Extending automatic parallelization to optimize high-level abstractions for multicore. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 28–41. Springer, 2009.

[195] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010.

[196] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman. Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[197] J. Lidman, D. J. Quinlan, C. Liao, S. McKee, et al. ROSE: Fttransform-a source-to-source translation framework for exascale fault-tolerance research. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[198] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[199] A. A. Lovelace. Sketch of the analytical engine invented by Charles Babbage by L. F. Menabrea of Turin, officer of the military engineers, with notes by the translator. *Bibliothèque Universelle de Genève*, 1843. Translation with notes on article in italian in Bibliothèque Universelle de Genève, October, 1842, Number 82.

[200] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski. Data-driven program completion. *arXiv preprint arXiv:1705.09042*, 2017.

[201] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski. Program splicing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 338–349. ACM, 2018.

[202] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In S.-C. Cheung, A. Orso, and M.-A. Storey, editors, 22^{nd} *International Symposium on Foundations*

of Software Engineering (FSE 2014), pages 643–653, Hong Kong, China, November 16 - 22 2014. ACM.

[203] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on API understanding and extended Boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 260–270. IEEE, 2015.

[204] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[205] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 45–52. IEEE, 1998.

[206] Z. Manna and R. Waldinger. A deductive approach to program synthesis. In *Readings in artificial intelligence and software engineering*, pages 3–34. Elsevier, 1986.

[207] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–164, 1971.

[208] K. Mao. *Multi-objective Search-based Mobile Testing*. PhD thesis, University College London, Department of Computer Science, CREST centre, 2017.

[209] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017.

[210] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 94–105, 2016.

[211] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 16–26, 2017.

[212] J. Maras, M. Štula, and I. Crnković. Towards specifying pragmatic software reuse. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 54. ACM, 2015.

[213] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[214] A. Marginean, E. T. Barr, M. Harman, and Y. Jia. Automated transplantation of call graph and layout features into Kate. In *Search-Based Software Engineering*, pages 262–268. Springer, 2015.

[215] V. Markovtsev and W. Long. Public git archive: a big code dataset for all. *arXiv preprint arXiv:1803.10144*, 2018.

[216] F. Martin, H. R. Nielson, C. Riva, and M. Schordan. Towards distributed memory parallel program analysis. *Scalable Program Analysis*, 2008.

[217] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[218] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[219] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[220] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *null*, page 260. IEEE, 2003.

[221] N. Meng, M. Kim, and K. S. McKinley. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 440–443. ACM, 2011.

[222] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *ACM SIGPLAN Notices*, volume 46, pages 329–342. ACM, 2011.

[223] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering (ICSE 2013)*, pages 502–511, Piscataway, NJ, USA, 2013.

[224] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

[225] T. Mens and T. Tourwe. A survey of software refactoring. *tse*, 30(2):126–139, Feb. 2004.

[226] C. Miles, A. Lakhotia, and A. Walenstein. In situ reuse of logically extracted functional components. *Journal in Computer Virology*, 8(3):73–84, 2012.

[227] J. F. Miller and P. Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer, 2000.

[228] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[229] D. J. Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

[230] Newsappweb. https://github.com/PowerMobileWeb/NewsApp, 2016. Accessed: 2020-01-24.

[231] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Grapacc: a graph-based pattern-oriented, context-sensitive code completion tool. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1407–1410. IEEE, 2012.

[232] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating code with statistical machine translation. In *ICSE*, pages 544–547. ACM, 2014.

[233] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: program repair via semantic analysis. In B. H. C. Cheng and K. Pohl, editors, *35th International Conference on Software Engineering (ICSE 2013)*, pages 772–781, San Francisco, USA, May 18-26 2013. IEEE.

[234] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden. Bamboo: translating mpi applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 39. IEEE Computer Society Press, 2012.

[235] P. O'Hearn. Separation logic. *Commun. ACM*, 2018. to appear. (will give web link in time).

[236] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, 2001.

[237] M. Orlov and M. Sipper. Flight of the FINCH through the java wilderness. *IEEE Transactions Evolutionary Computation*, 15(2):166–182, 2011.

[238] A. Ouaarab, B. Ahiod, and X.-S. Yang. Discrete cuckoo search algorithm for the travelling salesman problem. *Neural Computing and Applications*, 24(7-8):1659–1669, 2014.

[239] T. Panas. Signature visualization of software binaries. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 185–188. ACM, 2008.

[240] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating software architecture using a unified single-view visualization. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 217–228. IEEE, 2007.

[241] T. Panas and D. Quinlan. Techniques for software quality analysis of binaries: Applied to windows and linux. *DEFECTS*, 9:6–10, 2009.

[242] T. Panas, D. Quinlan, and R. Vuduc. Analyzing and visualizing whole program architectures. In *ICSE Workshop on Aerospace Software Engineering (AeroSE), Minneapolis, MN*, 2007.

[243] T. Panas, D. Quinlan, and R. Vuduc. Tool support for inspecting the code quality of hpc applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, page 2. IEEE Computer Society, 2007.

[244] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA '11)*, pages 199–209, New York, NY, USA, 2011. ACM.

[245] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *Proceedings of the* 22^{nd} *Symposium on Operating Systems Principles (SOSP'09), Operating Systems Review (OSR)*, pages 87–102, October 2009.

[246] L. E. Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[247] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2018.

[248] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014.

[249] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering*, 44(6):574–594, 2018.

[250] P. Pirkelbauer, C. Liao, T. Panas, and D. Quinlan. Runtime detection of c-style errors in upc code. In *Proceedings of fifth conference on partitioned global address space programming models, PGAS*, volume 11, 2011.

[251] D. A. Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, 2013.

[252] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

[253] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[254] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 107–126. ACM, 2015.

[255] pySchrodinger Developers. pyschrodinger. `https://github.com/jakevdp/pySchrodinger`, 2016. Accessed: 2016-01-12.

[256] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[257] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *ISSTA*, 2015.

[258] D. Quinlan, G. Barany, and T. Panas. Shared and distributed memory parallel security analysis of large-scale source code and binary applications. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2007.

[259] D. Quinlan and C. Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1, 2011.

[260] D. Quinlan, R. Vuduc, T. Panas, J. Härdtlein, and A. Sæbjørnsen. Support for whole-program analysis and the verification of the one-definition rule in c+. *Paul E. Black, Helen Gill, and W. Bradley Martin (co-chairs)*, 500:27, 2006.

[261] D. J. Quinlan, R. W. Vuduc, and G. Misherghi. Techniques for specifying bug patterns. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 27–35. ACM, 2007.

[262] S. M. F. Rahman, J. Guo, A. Bhat, C. Garcia, M. H. Sujon, Q. Yi, C. Liao, and D. Quinlan. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *Proceedings of the 9th conference on Computing Frontiers*, pages 123–132. ACM, 2012.

[263] O. Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[264] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel. Making malory behave maliciously: Targeted fuzzing of Android execution environments. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 300–311. IEEE, 2017.

[265] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 53. ACM, 2012.

[266] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Automated Software Engineering (ASE 2013))*, pages 367–377, Palo Alto, California, 2013.

[267] B. Ray, C. Wiley, and M. Kim. Repertoire: A cross-system porting analysis tool for forked software projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 8. ACM, 2012.

[268] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[269] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[270] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 81–90. IEEE, 2008.

[271] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan. Auto-scoping for openmp tasks. In *OpenMP in a Heterogeneous World*, pages 29–43. Springer, 2012.

[272] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608, 2015.

[273] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.

[274] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[275] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[276] K. Sastry and D. E. Goldberg. On extended compact genetic algorithm. In *Late-Breaking Paper at the Genetic and Evolutionary Computation Conference*, pages 352–359, 2000.

[277] A. S. Sayyad and H. Ammar. Pareto-optimal search-based software engineering (posbse): A literature survey. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on*, pages 21–27. IEEE, 2013.

[278] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[279] M. Schordan, P.-H. Lin, D. Quinlan, and L.-N. Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 493–508. Springer, 2014.

[280] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

[281] Scipy. https://www.scipy.org/, 2016. Accessed: 2020-01-24.

[282] J. Senellart, P. Dienes, and T. Varadi. New generation systran translation system. In *In Proceedings of MT Summit IIX Senellart J., Yang J., Rebollo A. 2003. SYSTRAN Intuitive Coding Technology. In Proceedings of MT Summit IX*. Citeseer, 2001.

[283] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitateporting to and across mobile devices and platforms. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 255–260. ACM, 2011.

[284] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, 2011.

[285] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: an extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 15. ACM, 2012.

[286] S. Sidiroglou-Douskos, E. Davis, and M. Rinard. Horizontal code transfer via program fracture and recombination. *CSAIL Technical Reports*, 2015.

[287] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 95–105. ACM, 2017.

[288] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard. Automatic error elimination by multi-application code transfer. In 36nd *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, Portland, Oregon, June 2015. To appear.

[289] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, pages 124–134, 2011.

[290] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12:1 – 12:48, June 2012.

[291] M. Simard, N. Ueffing, P. Isabelle, and R. Kuhn. Rule-based translation with statistical phrase-based post-editing. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 203–206. Association for Computational Linguistics, 2007.

[292] H. A. Simon. *The sciences of the artificial*. MIT press, 1996.

[293] P. Sitthi-amorn, N. Modly, W. Weimer, and J. Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152:1–152:11, 2011.

[294] A. Solar-Lezama and R. Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[295] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, volume 40, pages 281–294. ACM, 2005.

[296] Sorter Developers. Sorter. `https://github.com/Skinney/WordSorter`, 2016. Accessed: 2016-01-12.

[297] M. J. Sottile, C. E. Rasmussen, W. N. Weseloh, R. W. Robey, D. Quinlan, and J. Overbey. Foropencl: transformations exploiting array syntax in fortran for accelerator programming. *International Journal of Computational Science and Engineering*, 8(1):47–57, 2013.

[298] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *ACM SIGPLAN Notices*, volume 46, pages 492–503. ACM, 2011.

[299] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, volume 45, pages 313–326. ACM, 2010.

[300] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.

[301] J. E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[302] T. Su. Fsmdroid: guided GUI testing of Android apps. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 689–691. IEEE, 2016.

[303] J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-o-fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland, January 2014.

[304] T. swift-sorts community. swift-sorts. https://github.com/jessesquires/swift-sorts, 2016. Accessed: 2016-01-12.

[305] Equationsolver. https://github.com/nulLeeKH/quadratic-equation-solver, 2016. Accessed: 2020-01-24.

[306] Swiftsoupweb. https://github.com/scinfu/SwiftSoup, 2016. Accessed: 2020-01-24.

[307] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in android apps. In 40th *International Conference on Software Engineering (ICSE 2018)*, 2018.

[308] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury. Anti-patterns in search-based program repair. In 24th *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, pages 727–738, 2016.

[309] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *ACM Sigplan Notices*, volume 43, pages 295–312. ACM, 2008.

[310] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[311] G. Thurmair. Comparing rule-based and statistical mt output. In *The Workshop Programme*, page 5, 2004.

[312] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[313] V. Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[314] M. Trudel, C. A. Furia, M. Nordio, B. Meyer, and M. Oriol. C to oo translation: Beyond the easy stuff. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 19–28. IEEE, 2012.

[315] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

[316] A. Umbarkar and P. Sheth. Crossover operators in genetic algorithms: A review. *ICTACT journal on soft computing*, 6(1), 2015.

[317] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to design a program repair bot? insights from the repairnator project. In *40th International Conference on Software Engineering, Software Engineering in Practice track (ICSE 2018 SEIP track)*, pages 1–10, May 27 2018.

[318] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. Technical Report SEN-R9814, Centrum voor Wiskunde en Informatica (CWI), Sept. 1998.

[319] D. van Ravenzwaaij, P. Cassey, and S. D. Brown. A simple introduction to markov chain monte–carlo sampling. *Psychonomic bulletin & review*, 25(1):143–154, 2018.

[320] D. A. Van Veldhuizen and G. B. Lamont. Evolutionary computation and convergence to a pareto front. In *Late breaking papers at the genetic programming 1998 conference*, pages 221–228, 1998.

[321] D. A. Van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary computation*, 8(2):125–147, 2000.

[322] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[323] veusz Developers. veusz. https://github.com/jeremysanders/veusz, 2016. Accessed: 2016-01-12.

[324] R. Vuduc, M. Schulz, D. Quinlan, B. De Supinski, and A. Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 27–36. ACM, 2006.

[325] R. J. Waldinger and R. C. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252. Morgan Kaufmann Publishers Inc., 1969.

[326] P. Walsh and C. Ryan. Automatic conversion of programs from serial to parallel using genetic programming-the paragen system. In *PARCO*, pages 415–422. Citeseer, 1995.

[327] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 455–465, Saint Petersburg, Russian Federation, August 2013. ACM.

[328] W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 331–340. IEEE, 2014.

[329] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss. Hunter: next-generation code reuse for java. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 1028–1032. ACM, 2016.

[330] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss. Type-directed code reuse using integer linear programming. *arXiv preprint arXiv:1608.07745*, 2016.

[331] W. Weimer. Automatically finding patches using genetic programming (talk slides). $1^{st.}$ CREST Open Workshop, 24th - 25th November 2009, CREST Centre, London, UK.

[332] W. Weimer. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190. ACM, 2006.

[333] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[334] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009.

[335] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[336] M. Weiser. Program slicing. In 5^{th} *International Conference on Software Engineering*, pages 439–449, San Diego, CA, Mar. 1981.

[337] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[338] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation (TEVC)*, 15(4):515–538, 2011.

[339] Wikipedia. Lzf. https://en.wikibooks.org/wiki/Data_Compression/Dictionary_compression#LZF, 2016. Accessed: 2016-01-12.

[340] G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization—Eureka, You Shrink!*, pages 185–207. Springer, 2003.

[341] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-scale automated refactoring using clangmr. In *International Conference on Software Maintenance (ICSM 2013)*, pages 548–551, 2013.

[342] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.

[343] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[344] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 170–180. ACM, 2015.

[345] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *20th CAV*, pages 385–398, 2008.

[346] W. Yang, D. Kong, T. Xie, and C. A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, page 288–302. ACM, 2017.

[347] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 303–313. IEEE Press, 2015.

[348] K. Yarmosh. How much does an app cost: A massive review of pricing and other budget considerations. https://savvyapps.com/blog/how-much-does-app-cost-massive-review-pricing-budget-considerations, 2015. Accessed: 2018-01-20.

[349] S. Yoo. Embedding genetic improvement into programming languages. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, page 1551–1552. ACM, 2017.

[350] S. Yoo, X. Xie, F. Kuo, T. Y. Chen, and M. Harman. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 26(1):4:1–4:30, 2017.

[351] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126. IEEE, 2018.

[352] T. Zhang and M. Kim. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*, pages 665–676. IEEE Press, 2017.