Removing the Kitchen Sink from Software

Jason Landsborough Stephen Harding Sunny Fugate

SPAWAR Systems Center Pacific
San Diego, California, USA

ABSTRACT

We would all benefit if software were slimmer, thinner, and generally only did what we needed and nothing more. To this end, our research team has been exploring methods for removing unused and undesirable features from compiled programs. Our primary goal is to improve software security by removing rarely used features in order to decrease a program's attack surface. We describe two different approaches for "thinning" binary images of compiled programs. The first approach removes specific program features using dynamic tracing as a guide. This approach is safer than many alternatives, but is limited to removing code which is reachable in a trace when an undesirable feature is enabled. The second approach uses a genetic algorithm (GA) to mutate a program until a suitable variant is found. Our GA-based approach can potentially remove any code that is not strictly required for proper execution, but may break program semantics in unpredictable ways. We show results of these approaches on a simple program and real-world software and explore some of the implications for software security.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Tracing; H.4 [**Information Systems Applications**]: Miscellaneous; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program modification*

Keywords

Feature removal, tracing, genetic algorithm

1. INTRODUCTION

Early in the history of software, programs were carefully crafted to be as compact and efficient as possible. Space was only available at a premium. Ready availability of "extra features" has always been desirable, but constraints in early software ecosystems made this impractical. Today, most software is feature laden and built upon API stack upon API stack of prior effort, sometimes representing millions of lines of code. Unfortunately, today's software is often so overladen with superfluous features and latent "functionality" that even knowing what is included is a difficult question to answer.

1.1 Why thin

There is both an aesthetic desire and a security argument for whittling the required program to its bare minimum and nothing more. If we were to have an option between software which contained all possible features and "thinned" software, there are many possible benefits. We might make the following suppositions:

Pros:

- Security: Thinned programs do not contain the vulnerabilities present in code which was removed.
- Size: Thinned programs might be smaller and have performance gains over non-thinned code.
- Validity: Thinned programs are likely to be simpler and easier to validate, requiring fewer test cases.
- Optimality: Thinned programs can be better optimized for the specific machine, removing code for cross-compatibility.

Cons:

- Security: Thinned programs might not contain various security checks not normally raised in execution.
- Size: Thinned programs will fail or require additional time to load removed code segments.
- Validity: Thinned programs might require the same number of tests to validate that unwanted features are indeed removed.
- Optimality: Thinned programs may not run at all on other machines.

1.2 What to thin

We perceive two basic classes of program thinning: a) removing undesirable features; and b) removing unused features.

Removing undesirable features allows customization (via removal) of both intended functionality and unintended defects. By being focused on specific features, this approach limits adverse effects of code removal such as the unintended removal of security checks or error handling. This approach

This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.

GECCO '15, July 11 - 15, 2015, Madrid, Spain

ACM ISBN 978-1-4503-3488-4/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739482.2768424

is akin to a blacklist, where any feature on the blacklist is removed.

Removing unused features allows for discovery of program regions that are never executed. This can be thought of as "cold code" (as opposed to unreachable or "dead code"). Using this approach we can expect to remove large swaths of the program, but risk removing functionality or features related to security, safety, changing environments, configuration, or changing use-cases. This approach is akin to a whitelist, where any code executed during tracing is allowed, and all other code is potentially removed.

We might choose to thin a program for multifarious reasons, optimizing for any combination of: program size, code usage, feature desirability, vulnerability exposure, simplicity, or performance.

If a particular instruction is executed during normal use of a program, then we might first assume that these instructions were required. But a large portion of today's programs are dedicated to making software work in multiple environments. A significant portion of the program's code is dedicated to ensuring that regardless of which system call variants are used, or which version of libraries are available, that the program will still function as expected.

Similarly, programs with good exception handling will include a large quantity of code dedicated to handling error conditions and protecting against undefined behavior. Ad hoc removal of this code would be problematic at best, and catastrophic in some scenarios. When code is removed, it shouldn't be done arbitrarily and it should probably be done with utmost care. Currently this means that automated code removal must employ sophisticated knowledge of program semantics. An alternative approach could make error handling independent of the main program regions, ensuring that these regions could be ignored by automated code removal algorithms. Error handling code could also be marked or placed in reserved sections of the binary, making automated removal of other code sections trivial. We contend that for any particular computer system (to include the hardware, operating system, and configuration), there is a significant amount of redundant and unimportant code which could be removed without effect.

The English language is robust to the removal of unimportant words in sentences. Very recently, others in the field have begun exploring whether software shares a similar robustness [7, 5]. The robustness in human language appears to be primarily due to redundancy. We think the reason software is robust to code removal is due to safety. Software applications often include a large quantity of code ensuring safety and security properties which do not affect software functionality.

1.3 When To Thin

Different portions of a program can be most safely discarded at different times in the software life-cycle. The information available from source code is significantly different than information available when one only has a static binary, or when a binary resides on a target platform.

1.3.1 Source-code Thinning

Source-code thinning is, in essence, what any programmer might do when refactoring code to remove unused or unwanted features. A common manual method for thinning source code is to develop, maintain, and release from multiple independent branches of source code, each customized to specific use-cases. Source-based thinning is very similar to compile-time thinning through the use of compile-time directives. Automated removal of source code is complicated by the lack of information on how the software will be used in-practice. For this reason, software tends to be heavily laden with features that are used by only a subset of the user community. Adaptive forms of source code thinning are rarely used today, but could be easily employed by removing unused sections of source code based on debugging symbols and dynamic tracing. Other methods might use knowledge of programming language semantics to replace code sequences with more compact variants. These types of transforms are similar to those performed during optimization passes by the compiler.

1.3.2 Compile-time Thinning

Our work on feature removal is partially influenced by the infamous HEARTBLEED vulnerability. This vulnerability consisted of a logic error that leaked memory contents of servers using OpenSSL with the Heartbeat option enabled. It turned out that this feature was easily removed through the use of a compilation flag supplied during the build process, which was a recommended quick fix. Other forms of compile-time thinning occur when compiler directives (e.g. DEFINE and IFDEF) are used for "conditional compilation", selectively enabling or disabling features based on developer decisions, architecture, or availability of various libraries. This approach is commonly employed (and mostly safely used) by the developers of a program to enable useful variations in resulting binaries.

1.3.3 Binary Thinning

Customizing compile-time options for a program during compilation helps for the cases when source code is available. In many cases, particularly for those who use commercial software, the source code for programs is often not available. In these cases, modification through binary transformations may be required.

We focus on one method of binary transformations: binary thinning. Rather than run the default distribution of a program, we first perform dynamic analysis and then remove unwanted or unused features. This approach has been explored in some detail by others. Notably, Sethumadhavan et. al., Wagner et.al., and others have tested methods for removing cold code, dubbed "cold code removal" or "Binary Autotomy" [6, 8]

2. APPROACH

2.1 Manual Binary Thinning

First we examined the possibility of removing features from a program binary through static analysis and manually changing the bytes in the binary. The test program used was a small program written in C, that used a switch statement to control which function was called. Since this is an example that we want to keep simple and small, we omit good programming practices such as sanitizing inputs and bounds checking.

In our simple example, the feature that we want to remove from the compiled program in Figure 1 is contained within function 3, which is just a *printf* call. Many different approaches are possible, but here we only discuss two: func-

```
fun1(): print "This is function 1"
fun2(): print "This is function 2"
fun3(): print "This is function 3"
fundef(): print "Unsupported function"

main(arguments):
    if(arguments): option = get_argument()
    else: option = user_input("Enter option: ")

    switch(option):
        case 1: fun1(); break;
        case 2: fun2(); break;
        case 3: fun3(); break;
        default: fundef(); break;
```

Figure 1: Simple Program Pseudocode

tion overwriting and function redirection. The first method attempts to find the function (in machine code) which we want to remove, and overwrites all bytes except for the return statement in that function with NOP (x90) instructions, which do nothing. In this way, the function call can still be made but the function no longer performs the original operation. One advantage of a NOP overwrite method is that an exploit which relies on that function or a subset of instructions (code re-use attacks) would no longer be available for an attacker to use, providing some additional immunity to potential attacks.

Another straightforward method which can be equally effective is function redirection. This can be accomplished by modifying the operand of the call instruction to point to another function. Either method is easy to implement for trivial programs. The primary advantage of this approach is that it could ultimately be used to redirect a function call to some other function, such as one that notifies the user that the feature has been removed. Both of these approaches require disassembling a binary and understanding what feature needs to be removed, which gets increasingly difficult with larger and more complex programs. The program in Figure 1 is admittedly simplistic, but effectively demonstrates the basic principle of code removal.

2.2 Trace-based Binary Thinning

Decreasing the time and knowledge requirements of manual binary thinning requires some form of automated program analysis. A trace utility (such as distorm[1]) allows one to easily perform dynamic tracing of executed instructions, learning both the instruction offsets of unused (or undesired) features and instructions which need to be retained.

2.2.1 Prototype 1: Overwrite Unused

We created a prototype consisting of a set of scripts that disassemble a program and recorded all executable offsets and instructions to use as a baseline. The script then compares the output of the execution traces and marks recorded instructions as having been executed if they were executed at least once during the trace. This gives us the knowledge of which instructions were and were not executed, which we refer to as unused instructions. We are then able to overwrite instructions that were not executed by replacing them with no-operation instruction (NOPs).

Using this on our program from section 2.1, including traces of execution which collectively included fun1(), fun2(), and fundef(), we can easily remove the unused function fun3() by automating the process of overwriting with NOPs. This also removes instructions for the corresponding case in the switch statement. The resulting program uses the default case if someone attempts to use the removed feature. The original program contained 183 non-NOP instructions, and the thinned program using this method contained 133 non-NOP instructions.

In many cases a naive NOP overwriting approach will result in serious problems when users attempt to use a removed function, often resulting in a segmentation fault. Our simple program doesn't have any dependencies between function calls, so allowing the program to progress to the default function call has no deleterious effects. This naive approach may also remove both error handling code and any unused security checks.

One approach to making the approach more robust would be to use an exhaustive set of test cases, covering all possible error conditions and program inputs. However, it is often very difficult to guarantee that tests cover all possible edge cases or cover instructions that are executed in specific circumstances, such as during error conditions or those with possible timing dependencies.

2.2.2 Prototype 2: Overwrite Unwanted

To mitigate these issues, we decided to extend this approach, but be more selective in the instructions removed. This extended approach assumes the existence of a test case for the feature to be removed. Again, we want to remove the feature in fun3(). We now include a trace of execution including fun3(), and compare this trace with all the traces of functions we want to keep in order to find instructions unique to the function we want to remove, which we refer to as unwanted instructions. We tested this approach using the same binary we used in the previous method, which started with 183 non-NOP instructions, and the thinned program using this method, which contained 174 non-NOP instructions, with the same functionality we saw in section 2.2.1.

2.2.3 Echo Trace-thinned

We attempted this approach on a common program used by nearly every Linux user: `echo`. We selected an arbitrary feature to remove based on a cursory glance at the manpage. The `-n` option prevents echo from printing the trailing newline character, which it will print by default. We traced the execution of echo with a few options we wanted to keep, and traced the option we wanted to remove. Again, we took the difference of the traces with desired and unwanted features using the prototype script to overwrite these instructions unique to the unwanted feature. The original 32-bit echo binary contained 2960 non-NOP instructions, and the thinned binary contained 2958 non-NOP instructions. The instructions removed were a compare and a jump. As expected, attempting to use the program with the removed feature results in a segmentation fault. If the goal is just to remove a feature which will not be used, this result may be sufficient. In other instances, more robust approaches may be desired which guarantee program validity, performance, and security.

```
PROG_SIZE = length(program_variant)
PROG_NOPS = count_nops(program_variant)
FN = PROG_SIZE - PROG_NOPS
FITNESS = FN + RETVAL * 5
```

Figure 2: Fitness Calculation Pseudocode

2.3 Genetic Programming-based Thinning

Our next approach uses a genetic algorithm (GA) to modify a binary program in order to remove an undesired feature. Researchers have used genetic algorithms to modify programs in order to perform automatic program repair [2, 3] and optimization [4]. We extend this approach to automated feature removal (thinning) using the "Software Evolution" library used in [4]. The algorithm modifies programs through crossover, using tournament selection, and three available mutation operations on existing instructions: copy, delete, and swap. Genes are represented as the instructions within the binary. Crossover is two-point, where two offsets are selected such that the smaller program contains instructions at both offsets. The result of crossover contains the program from the start of the first program to the first offset, then the second program to the second offset, and then the rest of the first program. Our goal is to use the algorithm to mutate program binaries and reduce the number of instructions, either through removal of unnecessary instructions or transforming larger sets of instructions into fewer equivalent instructions.

The genetic programming process can terminate on either of two conditions: 1) the candidate solution has reached the desired fitness; 2) the number of evaluations has been reached or exceeded. We selected the second condition as our terminating condition. We also limit the population size to a maximum of of 512. To ensure we had sufficient resources for the genetic algorithm we used a 64-bit machine with 128 CPUs and 512 GB of RAM.

2.3.1 *Fitness*

Program variants' fitness is determined by a fitness script customized for the program to be thinned. This script consists of numerous test cases for features we want the program to retain, while the feature to be removed is simply excluded from these test cases. These test cases depend on comparing the output of the variant with what is expected. If a given variant in the population does not pass these tests, it is given a poor fitness value, which in this case is a very large value because we consider variants with lower fitness values to be better.

If all test cases pass, then, as shown in figure 2, we calculate a fitness value based on our goal. To prevent the file size from increasing, while also maximizing the number of removed instructions, our calculation is: file size - number of NOP instructions. Additionally, some solutions may still pass the test cases but return with a different value or produce abnormal behavior, such as executing normally but also resulting in a segmentation fault. To handle these cases, we adjust the previous calculation by multiplying the return value by 5, an arbitrary number we selected, and adding that to the initial fitness calculation. This allows candidate binaries in the population to remain competitive, but with a penalty.

2.3.2 *Simple program - GA Thinned*

Initially, we tested this approach on our simple program from section 2.1. Again, we retain the same goal of removing the functionality of fun3(). To accomplish this, we created a fitness script containing test cases, a total of 3, for the features we wanted to retain, where we check the program output against what we expect. If these test cases pass, we use the same fitness calculation shown in figure 2. The program started with a fitness of 7,433 with 180 non-NOP instructions. Here we examine the thinning results of the program after 300,000 evaluations. After the thinning process, the thinned binary had a fitness of 6,927 with 49 non-NOP instructions. All features we wanted still remained, and the removed feature resulted in "unsupported function" being displayed.

2.3.3 `echo` - *GA Thinned*

As with the trace-based thinning approach in section 2.2.3, we next attempted to thin `echo` by removing the `-n` option. We chose this program as a natural progression in complexity from our simple program. This time we started with the `echo` version available on our 64-bit machine. Since `echo` has more command line options than our simple program we decided to automate the test case generation. Within our fitness script, we randomly generated command line arguments for `echo` that randomly selected zero or more command line options, and randomly generated a string. We removed any bash-specific special characters, such as the semicolon, and generated a new test case for any that contain the option we are trying to remove: `-n`. To ensure good coverage in test cases for each fitness evaluation, we randomly generate 1000 of these test cases every time the script is run (each fitness evaluation).

Before thinning, `echo` had 2,609 non-NOP instructions. Using the same fitness calculation discussed earlier, `echo` started with a fitness of 28,056. After 300,000 fitness evaluations by the GA, the resulting thinned `echo` had 2,173 non-NOP instructions and a fitness of 25,714. This new version of `echo` supports all the features we wanted, and instead of a segmentation fault, which we expected, `echo` ignores the `-n` option and prints the rest of the parameters as it normally would, including the trailing newline. This result is better than what we achieved with trace-based thinning on `echo`.

2.3.4 `sha1sum` - *GA Thinned*

Next we wanted to try this approach on a more complex program. For this we chose `sha1sum`, which uses a sophisticated cryptographic algorithm, and chose the version available on our 64-bit machine. In this case we chose to only keep the sha1 checksums on individual files and standard input as our desired features. While we only have two features we care about, we went with a randomized approach again. The first set of test cases consists of a random string piped to `sha1sum` as standard input, repeated with a different random string 50 times. The next set of test cases consisted of running `sha1sum` on a random file containing random bytes, repeated with a different random file 50 times.

Before thinning, `sha1sum` had 5,049 non-NOP instructions. Using the same fitness calculation used with other thinning runs, `sha1sum` started with a fitness of 39,790. After 300,000 fitness evaluations by the GA, the resulting thinned `sha1sum` has 4,764 non-NOP instructions and a fitness of

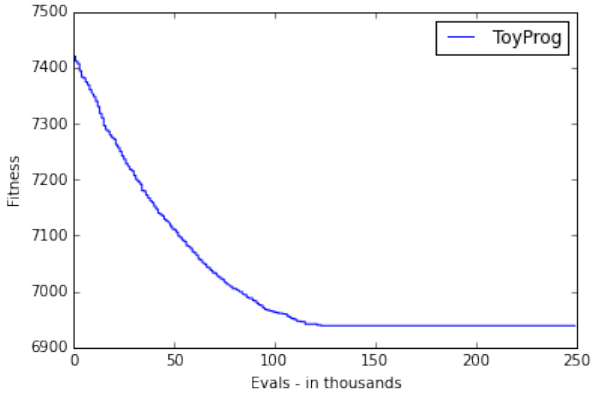

Figure 3: SimpleProg Fitness Plot

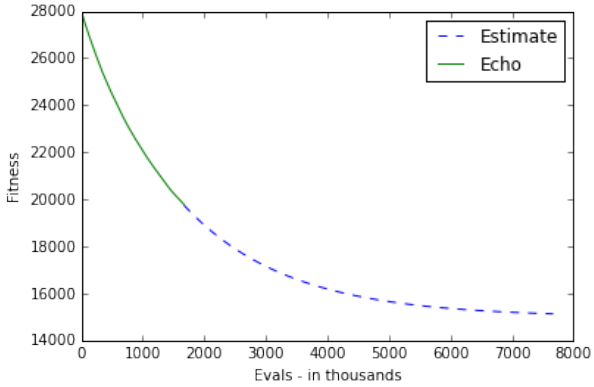

Figure 4: Echo Fitness Plot

37,787. The new version retains support for the two features we wanted.

2.3.5 How Low Can We Go?

We arbitrarily selected 300,000 evaluations for thinning our selected programs in the sections above. Here we explore the fitness of thinned programs evolved for a much longer period of time. There will eventually reach a point when no more instructions can be removed from the binary and have it still pass our test cases. What we did not know is whether this occurs abruptly or gradually by approaching some asymptote.

The simple program started out with the fewest instructions, and as a result we expect it to reach its limit the quickest. As seen in figure 3, the fitness roughly follows an exponential decay curve and has leveled off around 125,000 fitness evaluations.

As of this writing, the thinning of `echo` is still running and has not quite reached 2 million fitness evaluations. Using the results we already have, as shown by the solid line in figure 4, we can see that the fitness is not decreasing linearly anymore. If we fit a curve to the data we already have and keep plotting, we get the estimate curve as shown by the dashed line. Here we can see that `echo` may level off somewhere after 7 million fitness evaluations (3 million evaluations after our hard-coded limit of 4 million) with a fitness around 15,000.

2.3.6 Minimized Changes

Removing as many instructions as we can is arguably risky. The instructions that remain are entirely dependent on the robustness of the test cases used in the fitness script. If the test cases don't cover some edge case or error case, the instructions that handle these cases will eventually be removed. If it is only desired to remove the target feature and nothing else, there are some additional methods that can be used.

The first is the simplest case. If the target feature can be identified, it can be used to trigger the fitness termination condition of the GA. If we set the GA to terminate when a fitness of zero has been reached, we can then modify our fitness script to return zero when we identify that the feature has been removed. When we tried this approach with `echo` to remove the `-n` option, the GA has produced a binary with the feature removed in as few as 706 fitness evaluations. This resulting binary contained 2,607 non-NOP instructions, two fewer instructions than when we started.

Additionally, we can reverse changes to the binary using delta-debugging [9], to get the minimal set of changes necessary to remove the feature. If we supply the delta-debugging tool with the original binary, the thinned binary, a target fitness (zero in this case, where the feature has been removed), and the fitness script, it will undo changes to the binary instructions. The resulting binary will contain the minimum number of changed instructions, without the `-n` option in our case. After this process the resulting binary contained 2,608 non-NOP instructions, one fewer instruction than the original `echo`.

2.3.7 Current Issues and Limitations

Machine-Dependent: We have observed that given enough time spent thinning, the GA will strip out extra code that will allow the elf binary to run on many Linux systems, and therefore become machine-dependent. There may be cases where this is a desired result. In cases where this is not, it may be necessary to consider this when constructing test cases, or penalize variants that remove such code.

No GUIs: Our approach and fitness script does not currently support GUI-only based programs, as we are comparing standard output for the test cases. Additionally, it may be difficult to run multiple large GUI-based programs in parallel, and some may specifically prevent it.

Large Programs: Using a GA with significantly larger programs may suffer a proportional increase in the number of evaluations needed to obtain desired fitness. We believe this is unlikely for common applications as these programs also contain a proportionally larger number of undesirable or unnecessary features. The fitness goal may also be designed to remove only a fixed number of instructions.

Test-Suite Dependence: Our approach is entirely dependent on the quality of the test-suite used to verify a program's correctness, performance, and security. For many programs, test suites only test a small portion of the underlying code. When code-coverage is poor, these untested features are likely to be removed. The consequences may be programs which are robust only when run with identical data inputs, configuration, environment, and user interactions. We believe this limitation suggests that programs should have broader-scoped and more complete test suites.

Time: While the fitness evaluations can be highly parallelized, because the transformations are random and frequently lead to failures, GA-based thinning can take a lot of time.

3. FUTURE WORK

Binary Diversity: In addition to any additional security we might get through a reduction in attack surface, we suspect that there may be additional benefits using the GA approach with binary diversity when instructions are swapped or sets of instructions are reduced to a smaller set. Also, if we examine the types of valid instruction transformations, we may be able to discover useful binary transformations that can be used independently of the GA approach, such as after trace-based thinning.

Performance: There may be utility in factoring in performance with the fitness value. We suspect this could bias solutions where executed instructions are grouped in lower memory addresses, whereas removed instructions replaced with NOPs are pushed to higher memory addresses and may not be executed at all if placed after a return or jump. Similar work has been done for energy usage by Shulte, et. al. [4].

Failing Gracefully: In most cases, when the removed feature is used, it results in a segmentation fault. It would be desirable to either exit without errors, or redirect to some other code such as a message that notifies the user that the feature is unknown or has been removed.

Simplifying Reverse Engineering: We suspect the GA-approach could be used to help simplify reverse engineering. For example, by mutating a piece of software such as malware, we may be able to unpack that malware, or remove any anti-reverse engineering mechanisms. This, however, assumes a sophisticated set of tests can be generated for the malware ahead of time.

4. CONCLUSION

We have described two different approaches for feature removal in program binaries. The first approach uses a trace of program execution as a guide, which was further refined by explicitly removing unique instructions found in a trace of the program using the feature to remove. The second uses a genetic algorithm to mutate a program, eventually removing the undesired feature. The changes made by the genetic algorithm are random and likely risky, but can be reduced by undoing changes to the minimal set needed to remove the feature. We showed that instructions can be removed post-compilation in binaries to remove unwanted features, while maintaining the functionality of features we want to retain. In both cases, if an exploit required the existence of the removed feature, it would not likely work. These approaches are promising, and if they continue to improve, one day we may have the software we want, rather than the software we get.

5. ACKNOWLEDGEMENTS

The authors are grateful to Jeff Knockel and Antonio Espinoza at the University of New Mexico for supplying us with the trace tool we used in the trace-based thinning approach. We would also like to thank Mark Bilinski, Daniel Gebhardt, and Geancarlo Palavicini for reviewing this paper.

6. REFERENCES

[1] G. Dabah. distorm - powerful disassembler library for x86/amd64. URL: https://code.google.com/p/distorm/, January 2015.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, Jan. 2012.

[3] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 317–328, New York, NY, USA, 2013. ACM.

[4] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. *SIGARCH Comput. Archit. News*, 42(1):639–652, Feb. 2014.

[5] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, Sept. 2014.

[6] S. Sethumadhavan, S. J. Stolfo, A. Keromytis, J. Yang, and D. August. The sparchs project: Hardware support for software security. In *Proceedings of the 2011 First SysSec Workshop*, SYSSEC '11, pages 119–122, Washington, DC, USA, 2011. IEEE Computer Society.

[7] M. Velez, D. Qiu, Y. Zhou, E. T. Barr, and Z. Su. A study of "wheat" and "chaff" in source code. *CoRR*, abs/1502.01410, 2015.

[8] G. Wagner, A. Gal, and M. Franz. "Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading. *Science of Computer Programming*, 76(11):1037–1053, Nov. 2011.

[9] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.