Software Security through Targeted Diversification

Nessim Kisserli Jan Cappaert Bart Preneel

Katholieke Universiteit Leuven, Dept. of Electrical Engineering – ESAT Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium
{nessim.kisserli,jan.cappaert,bart.preneel}@esat.kuleuven.be

***Abstract*— Despite current software protection techniques, applications are still analysed, tampered with, and abused on a large scale. Crackers[1] compensate for each new protection technique by adapting their analysis and tampering tools. This paper presents a low-cost mechanism to effectively protect software against *global* tampering attacks. By introducing diversity *per programme instance*, we illustrate how to defeat various patching methods using inlined code *snippets*. We propose an efficient technique for creating the snippets based on genetic programming ideas, and illustrate how our approach might trigger a small-scale arms race between defending and attacking parties, each forced to evolve in order to "stay in the game".**

I. Introduction

Piracy has plagued software vendors for years and still continues to do so. Efforts to thwart it have largely failed due to the inherently open architecture of current computing systems and the prevalence of a healthy software monoculture. Attempts to address the former can be seen in the current development of Trusted Protection Modules (TPMs), the latter impediment however, despite being acknowledged [10], remains mostly unaddressed.

Table I illustrates the cracking process' various stages as described in [12] and the typical countermeasures employed by software vendors to counter each. We note the lack of any widespread protection mechanism targeting the automation stage and propose a low-overhead solution based on the following sober reflections:

- Users will always attempt to analyse software protection mechanisms, out of academic curiosity or otherwise.
- It is virtually impossible to prevent a user from modifying software and processes on a machine they control –given current architectures.
- It is impossible to police the Internet and remove the myriad sites currently hosting pirated software.

Software patches contain sufficient information to locate and replace a set of *critical instructions* within a programme. We propose to make automated patching sufficiently unreliable, and consider our approach successful if crackers:

- Are forced to distribute cracked full-binaries, or
- Develop automated patches whose size approaches that of the full-binary.

[1]We use the term for individuals who strip copy-protection mechanisms from commercial software, enabling its unfettered use. They are also colloquially referred to in some circles as *warez d00dz* [13]

Both cases cause sites hosting pirated software to incur substantially higher bandwidth and storage costs. Additionally, the former allows software developers to leverage watermarking and other origin identification techniques such as [16].

This paper is structured as follows. Section II introduces *snippets*, the building blocks of our protection scheme. Section III explores increasingly sophisticated existing as well as hypothetical automated patching techniques, describing specialised snippets to counter each. We share our experiences generating such snippets using genetic programming in Section IV, and present related research and ideas for further work in sections V and VI respectively.

II. Snippets

A snippet is a series of one or more assembly instructions designed to be inserted within an existing assembly programme which we refer to as *the host*. We distinguish three types of snippets based on the net effect of their execution on a host's state. Informally, two instances of a programme are said to be in the same state if the following conditions hold:

- Their instruction pointers refer to the same instruction.
- Their stack pointers refer to the same stack offset.
- The contents of their respective registers are identical, including the flags register.

A snippet is termed *harmless* if executing it in a host at state σ_1 yields a new state σ_2, identical to σ_1. Intuitively such snippets exhibit similar properties to redundant code. Non-harmless snippets are further distinguished into semi-harmless and harmful ones. A *Semi-harmless* snippet is one which under certain well defined conditions, we call *insertion conditions*, can be rendered harmless. The task of locating in a host *insertion points* at which a snippet's insertion conditions are satisfied is carried out by an *insertion function*. Snippets with a non-empty set of insertion conditions for which no insertion function can be found are *harmful*.

A. Immunity from code compaction

Being extraneous code, we must ensure the snippets are not readily identified using code compaction tools. There is a paucity of work on code compaction of compiled, stripped binaries as most available research focuses on compiler-generated parse trees with full access to authoritative source code, e.g. [5]. Our snippets can take the following forms:

- Redundant code: Code whose execution has no effect on the overall output of a programme.

TABLE I
THE CRACKING PROCESS AND TYPICAL COUNTERMEASURES

Stage	**Purpose**	**Industry adopted protection mechanism**
Analysis	Determine and locate protection mechanisms	Obfuscation
Tampering	Disable protection mechanisms	Tamper resistance
Automation	Apply the tampering to other instances of the software with minimal user knowledge and interaction	None
Distribution	Provide others with the ability to obtain cracked software	Legal threats + Termination of hosting

- Dead code: Code whose computed results are unused.
- Unreachable code: Code to which there is no control flow path. Determining whether an arbitrary code snippet is reachable is considered undecidable.

The use of various obfuscation concepts in our snippets (such as opaque predicates [3] for branch-confidentiality) can help reduce the threat of detection. While the remainder of this paper classifies snippets according to various criteria, we assume them all immune to detection by code compaction tools.

III. THE ARMS RACE

While the battle lines between crackers and software vendors are better defined in the *analysis* phase of the cracking process, they are no less present in the *automation* stage. In this section we systematically examine automated patching techniques in order of increasing complexity. For each method, we discuss required properties of both snippets and insertion functions to counter automation, further escalating the arms race.

As we approach the limits of sequential instruction-based comparisons, we explore more "exotic" methods such as graph-based structural programme analysis. While the patching methods in sections III-A and III-B are widely used by the cracking community, those outlined in sections III-C and III-D are, to the best of our knowledge, currently not.

A. Offset patches

1) How they work: As their name suggests, offset patches overwrite a number of bytes at a fixed file offset. They may employ various techniques to maximise successful patching, such as comparing the binary's checksum against a "known good value" or verifying the instruction at the specified offset is the expected one. Their continued successful use by crackers is testimony to the inherent weakness of today's software monoculture on the one hand, and the failure of current protection schemes to address all stages of the cracking process on the other.

2) Defeating them: Any modification to the offset of the critical instructions will suffice to defeat an offset patch. The main requirement is that the snippet be inserted before the critical instructions. Such snippets, which need not exhibit any special properties, we call *basic snippets*, and are in fact a superset of the more specialised harmless snippets introduced later.

B. Pattern searching patches

1) How they work: Pattern searching patches locate and replace specific byte patterns in a binary. They may also employ similar success maximising techniques to offset patches.

2) Defeating them: There are two main approaches to defeating pattern matching cracks:

- Destroy the pattern being searched for.
- Duplicate the pattern, inducing multiple false positives.

The first technique requires interleaving snippets with critical instructions. In the most extreme case, no two consecutive native instructions are allowed to remain in the critical section. Harmless snippets, by definition, can be inserted between any two instructions without disrupting the host's functionality. A targeted insertion function is required for inserting the snippets between the critical section's instructions. We call such snippets which actively destroy patterns in their host *parasitic snippets*.

The second technique requires the creation of snippets containing sequences of instructions identical to the critical ones. However, these instructions are most likely to be harmful and must first be "neutralised". Use of opaque predicates can guarantee such instructions are always skipped (i.e. rendered dead code). We call these snippets which imitate their host *mimic snippets*.

We note that both approaches can be employed simultaneously for increased effectiveness.

C. Collusion-based patches

1) How they work: Collusion attacks refer to ones in which multiple parties share information about a protection scheme in order to defeat it. At its simplest, such an attack takes the form of a file comparison between two or more diversified instances of a programme to establish their commonality.

2) Defeating them: We propose a new kind of snippet for defeating such collusion attacks, the *poisoned snippet*, with the following properties.

- It is composed of two consecutive logical parts S_1 and S_2. Crucially, when combined they form the harmless snippet $S = S_1 \| S_2$ (where $\|$ denotes concatenation).
- Taken separately, both parts are most likely harmful.
- A *generation* of poisoned snippets all share the exact same logical part S_x.

Poisoned snippets require the following insertion condition:

- All snippets in a generation are inserted adjacently to the same native instruction (we call a *border instruction*) in all instances of the diversified programme.

The above condition effectively renders the most likely harmful S_x–part of the poisoned snippet indistinguishable from native instructions by including itself in the Longest Common Substring (LCS) spanning the border instruction.

D. Structural Analysis-based patches

1) How they work: Comparative structural analysis is generally cast as a graph matching optimisation problem. Two differing but similar executables E_1 and E_2 are represented as graphs G_1 and G_2 and an optimal isomorphism between the two is sought.

Nodes from each graph representing the same element in E_1 and E_2, called *fixed points* in [6], are used to map each function in E_1 onto its counterpart in E_2 using function signatures (such as return, number and type of formal parameters). The same procedure is iterated for each function's basic blocks, then for each basic block's individual instructions until a partial best-fit isomorphism between G_1 and G_2 is achieved. The resulting bijection maps elements of E_1 to their semantic equivalents in E_2. Recently both [6] and [15] have used structural analysis to highlight changes between different patch-levels of a binary.

It is still not clear whether this type of patch will be feasible. It incurs higher storage and bandwidth usage to access the graph of a known-good patched instance, and may be less economic than simply distributing a fully cracked binary.

2) Defeating them: Structural analysis based patching can be attacked in two ways:

- Obfuscating a programme's control and data flow, complicating initial graph construction.
- Minimising iteratively discoverable fixed points across programme instances.

The first approach can leverage existing obfuscation and anti-disassembly techniques. The latter requires the creation of *decoy snippets* providing fake fixed points upon which graphs are mapped onto each other. They follow the same principle as mimic snippets, but are structurally more complex as they must mimic both function signatures and control flow properties. To be effective, the insertion function must be coupled with targeted modifications to the host's original code. This is what we propose to explore in more detail in the diversifying compiler mentioned in section VI.

IV. Snippet generation

Manually crafting snippets for our protection scheme presents a costly endeavour given their required number and specificity. Rather, we rely on genetic programming techniques to automate the task, finding the stochastic and evolutionary elements of the approach particularly appropriate.

In this section we share some of our experiences evolving various aspects of the snippet species we introduced in section III. We discuss our main fitness function, different *genetic operators* used, and illustrate some of the problems faced. Due to space constraints, we assume familiarity with the basic workings of a genetic algorithm.

A. Genetic operators

Genetic operators are the main drivers of diversification in genetic computing and are broadly divided into so called sexual and asexual types. The former traditionally combine traits from two parents to produce an offspring, while the latter mutate one individual into another.

a) Mutation: Asexual reproduction was limited in our experiments to infrequent mutations in which one of the following occurred:

- Two random instructions in a snippet were swapped.
- A register was substituted for another, globally within a snippet. For example, all references to **eax** changed to **ebx**.

The latter form was introduced in an attempt to curb the observed general destructiveness of the first mutation, and to allow for a limited template-like replication of harmless individuals.

b) Reproduction: We experimented with two operators, the classical *N-point crossover* and a more suitable, snippet-friendly *Insertion* operator. The former divides two snippets X and Y into *n* random parts $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ respectively, recombining them into two new child snippets $C_1 = X_1 \| Y_2 \| \ldots X_{n-1} \| Y_n$ and $C_2 = Y_1 \| X_2 \| \ldots Y_{n-1} \| X_n$. The operator was found to be extremely destructive in the majority of cases, including $n = 2$.

Starting from the observation that harmless snippets can be safely embedded between any two instructions, we developed the less destructive *insertion operator*. Here, one of the two parent snippets is chosen and randomly split into Y_1 and Y_2. The remaining parent, X, is then inserted between the two parts, creating a new child snippet C=$Y_1 \| X \| Y_2$.

While particularly well suited at duplicating mimic snippets and overall less destructive, the insertion operator produces increasingly longer snippets which must be artificially constrained.

B. Snippet Simulation

In order to establish the effect of snippets on a host programme's state, we model various IA32 assembly instructions and a generic x86 little-endian compatible execution environment as follows:

- Several general purpose 32-bit registers modelled with bit-level precision.
- An extended 32-bit flags register ϕ.
- A programme stack and accompanying stack pointer ρ.

We initialise our pre-snippet execution environment σ_1 as follows:

- For each bit *i* in the extended flags register, set $\phi[i] = 0$.
- For each simulated register, *reg* set $\sigma_1(reg)[] = reg[31], reg[30], \ldots, reg[0]$.
- Initialise the stack's pointer ρ_{σ_1} to 0.

We then symbolically evaluate, with bit-level precision, the effect of each assembly instruction on our environment. While tracking changes to the stack is relatively straight forward, we rely on a bit vector *decision procedure* for the individual bits of each register (we use the function rich STP [9] from Stanford university).

At the end of a snippet's execution our environment is in state σ_2. Recall from section II that a harmless snippet is one which does not modify its original environment, i.e. one for which $\sigma_1 == \sigma_2$. Practically, for a harmless snippet:

- The stack pointer $\rho_{\sigma_2} == 0$.
- For each simulated register, *reg*, the symbolic value $\sigma_2(reg)[i] == \sigma_1(reg)[i]$ for i=0 to 31.

C. Flags

Up to now we have carefully avoided mentioning the flags register. This is the most problematic aspect of symbolic evaluation. The issue can be sidestepped by restricting ourselves to only modelling those instructions which do not set any flags, such as **push** and **pop**. This is not a practical solution however. Our current approach adds an appropriate restriction to the snippet's insertion condition set for each simulated instruction which may trigger a flag. Most such snippets are semi-harmless, requiring an adequate insertion function. However, certain snippets which make use of constant values or opaque predicates may escape such restrictions if the simulator is able to assert flag setting-conditions are not met.

1) Insertion Conditions: The following fragment for example, risks overwriting the Zero Flag (ZF) upon equality and the Carry Flag (CF) if $ebx > eax$:[2]

```
cmp eax ebx
jnz L1
...
```

The insertion conditions are thus:

- $\phi[ZF]$ is not live.
- $\phi[CF]$ is not live.

This is most likely the case right before a native **cmp** instruction.

V. Related research

Diversification has been leveraged in many security solutions. Cohen explored the feasibility of using similar techniques to ours to increase operating systems' resistance to attacks in [2]. Forrest et al. sketched a multi-level holistic system diversification technique, including instruction block reordering and use of nonfunctional code in binaries [8]. More recently, Anckaert et al. discussed the logistics of providing updates to tailored software instances [1], and although conceptually similar to our idea, no details of the tailoring scheme were provided.

More generally, Address Space Layout Randomisation [14] (ASLR), System Call diversification, and Instruction Set Emulation have all been used as countermeasures to certain types of memory corruption and code-injection attacks. Cox et al. formalised the use of diversification and equivalent execution in their N-variant system [4], targeting similar classes of attacks. Finally, El-Khalil et al. used functionally equivalent instructions for steganographic purposes [7].

[2]For simplicity we assume unsigned operands.

VI. Conclusions and further work

Besides modelling additional assembly instructions and exploring new opaque predicates, the problem of the flags register must be better addressed. We would like to incorporate our snippets into a diversifying compiler in order to influence and better exploit the layout of the native assembly. By producing keyed one-to-many mappings between the high-level *native instructions* and their assembly, snippets can be made harder to distinguish. Currently we lack automated insertion functions. A compiler lends itself fairly naturally to this task. Adding such functionality to the LLVM compiler framework [11] is therefore our next goal.

This paper introduced programme diversifying snippets, a light-weight software protection scheme designed to thwart patches relied on for low-overhead, mass distribution of pirated software. We presented several types of snippets targeting distinct patching approaches, and showed the feasibility of automating their creation using genetic programming techniques.

References

[1] B. Anckaert, B. De Sutter, and K. De Bosschere. Software Piracy Prevention Through Diversity. In *Proceedings of the 4th ACM workshop on Digital rights management* pp. 63–71, Washington DC, 2004.

[2] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):56–584, 1993.

[3] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages 1998*, San Diego, CA, January 1998.

[4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *The 15th USENIX Security Symposium*, pp. 10–120, 2006.

[5] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Language and Systems*, Ed. 22(2), March 2000.

[6] T. Dullien and R. Rolles. Graph-based comparision of executable objects. *Symposium sur la Securite des Technologies de l'Information et des Communications*, 2005.

[7] R. El Khalil and A. D. Keromytis. Hydan: Hiding Information Binaries. In *Proceedings of the 6th International Conference on Information and Communications Security*, pp. 187–199, October 2004, Malaga, Spain.

[8] S. Forrest, A. Somayaji, and D. H. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pp. 67–72, 1997.

[9] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. *Computer Aided Verification*, Berlin, Germany, July 2007.

[10] D. Geer *et al.* CyberInsecurity: The Cost of Monopoly, 2003.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, Palo Alto, CA, 2004.

[12] A. Main and P.C. van Oorschot. Software Protection and Application Security: Understanding the Battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography*, Heverlee, Belgium, June 2003.

[13] E. Raymond(ed), The New Hacker's Dictionary, MIT Press, 1991.

[14] The PaX Team. `http://pax.grsecurity.net/docs/aslr.txt`

[15] T. Sabin. Comparing binaries with graph isomorphisms. *BindView RAZOR Team*, 2004. `http://www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm`

[16] Julien P. Stern, G. Hachez, F. Koeune, and J. Quisquater. Robust Object Watermarking: Application to Code. *Information Hiding '99, volume 1768 of Lectures Notes in Computer Science* pp. 368–378, Dresden, Germany, 2000.