Unconditional self-modifying code elimination with dynamic compiler optimizations

Isabelle Gnaedig, Matthieu Kaczmarek, Daniel Reynaud, Stéphane Wloka

Unconditional self-modifying code elimination with dynamic compiler optimizations

Isabelle Gnaedig
INRIA - LORIA
Isabelle.Gnaedig@loria.fr

Matthieu Kaczmarek
INRIA - LORIA
matthieu.kaczmarek@inria.fr

Daniel Reynaud
Nancy Université - LORIA
reynaudd@loria.fr

Stéphane Wloka
Nancy Université - LORIA - ENSIMAG
stephane.wloka@ensimag.fr

Abstract

This paper deals with the issue of self-modifying code and packed programs, a long-standing problem commonly addressed by emulation techniques and memory dumps. We propose an original semantics-based approach to simplify dynamic code analysis, by using compiler optimization techniques to get rid of code-generating instructions. For this, we use classic slicing techniques to identify code dependencies. As it is semantics-based, our approach allows us to rely on strongly established formal methods and is a promising approach for handling packed programs.

1 Introduction

Packed malware is a long-standing problem which appeared very early in the history of computer viruses. Until relatively recently, the state of the art in unpacking technology consisted in a set of specific unpacking procedures created manually for known packers [22]. The limit of this reactive approach has been reached in the early 2000s, due to a significant increase in the number of self-modifying malware samples and custom packers. Since then, generic unpacking and emulation techniques have constituted a hot topic in the malware community.

Related works. Let us briefly review studies that tackle the problem of generic unpacking.

PolyUnpack [19] and VxStripper [11] compare the binary code and its image in memory in order to detect runtime-generated instructions.

Renovo [12] and Pandora's Bochs [2] both detect the presence of unpacked code by monitoring whether the instruction stream reaches a previously written memory area. This is achieved through whole system emulation. The approach of Ether [6] is similar, but the hardware virtualization extensions of modern CPUs are used instead of emulation.

In [21], Hump-and-dump detects dynamic code by using execution histograms.

OmniUnpack [14] makes use of the virtual address translation feature of CPUs in order to track the modifications in memory and identify packed code. The heuristics combine static and dynamic analysis to monitor memory writes.

Justin [9] employs the *no execute* bit of CPUs in order to take control and trigger a malware scan when a data region is executed.

In [4], the authors propose an efficient layering algorithm based on an emulation framework for user-mode x86 and the Windows API.

The main limitation of these approaches based on memory monitoring is that the tools are not guaranteed to output valid executables. Indeed, the tools attempting to reconstruct an unpacked binary generally rely on a memory dump, repre-

senting a particular state of the program at a given point in time. This representation lacks semantics information, so the reconstructed binary may not be semantically equivalent to the original program or even outright broken [7], for instance due to code erased from memory or not yet decoded.

Moreover, the techniques used to monitor execution generally incur a significant performance overhead. As a result, generic unpacking techniques are either not deployed on end-user desktops or are deployed as watered-down versions. Binaries to be unpacked are rather submitted to dedicated sandbox analysis services.

Contribution. In this paper, we propose a fundamentally different approach to extract and restore hidden code, called program flattening. It can be summed up as follows: if we consider that a packed program is the association of an encoded payload and an unpacking stub that *unconditionally* restores the original program by self-modification, then observing any trace of the packed program is sufficient in order to *recover the generated code* and to *eliminate code-generating instructions* with a data dependency analysis.

This approach differs from previous works by the use of program transformations based on semantic information: data flows are computed based on x86 instruction semantics. Moreover, this perspective allows to rely on established formal methods whereas previous approaches usually employ black box analysis. Another contribution of our work is the relation that we draw between program optimization and unpacking.[1]

The paper is organized as follows. Section 2 details the theory behind program flattening and explains how it can be applied to packed programs. Section 3 presents our optimizing technique on execution traces and illustrates its behavior on a simple but realistic self-modifying x86 program. Section 4 discusses the limitations of the approach. Finally, Section 5 presents the next steps of this research vein.

[1]Its development is however in a preliminary stage, therefore it has not been tested on malware samples at the time of writing.

2 Program flattening

2.1 Futamura's projection

An interpreter is a program with an argument x such that launched with $x = \mathbf{p}$, it simply executes the program $\mathbf{p}$. In the Linux world, `bash`, `perl` and `python` are well-known interpreters.

Specialization is a program transformation which turns programs with $n + 1$ arguments into programs with n arguments. This process is achieved by a program **spec**, commonly referred to as a specializer or a partial evaluator. Nowadays this notion is well-known since it constitutes the basis of most JIT compilers. A good introduction to program specialization for performance can be found in [10].

The combination of an interpreter and a specializer is also well-known in compilation theory since they constitute fundamental building blocks of compilers. Indeed, the Futamura projection [8] tells that a compiler $\mathbf{cc}_T^S$ from a source programming language S to a target machine code T can be obtained by combining an interpreter of S programs written in T, noted $\mathbf{exec}_T^S$, and a specializer of T, noted $\mathbf{spec}_T$.

$$\mathbf{cc}_T^S(\mathbf{p}) = \mathbf{spec}_T(\mathbf{exec}_T^S, \mathbf{p})$$

With $\mathbf{p}$ a source program written in S, executing a target program $\mathbf{t} = \mathbf{cc}_T^S(\mathbf{p})$ is by definition of $\mathbf{exec}_T^S$ semantically equivalent to executing program $\mathbf{p}$. From the above equality, we observe that $\mathbf{t}$ is the machine code program resulting from the compilation of the source program $\mathbf{p}$.

2.2 Packing as factitious compilation

We now draw a relation between compilation and packing. In their simplest form, packers are composed of an encoded payload and a header program which decodes and transfers control to the payload. The payload can either be compressed or encrypted, depending on the design goal of the packer: code size reduction or obfuscation. UPX[2] is a well-known packer used for code compression

[2]`http://upx.sourceforge.net/`

that conforms to this model. In other terms, a packed program is an encoded program wrapped into a runtime decoder.

From a theoretical perspective, packing can be seen as a particular compilation process: a packer is seen as an interpreter and the payload is seen as the source program. Let us consider a symmetric transformation process with encoding (resp. decoding) procedure **encode** (resp. **decode**). That is, for any data a we have

$$\mathbf{decode}(\mathbf{encode}(a)) = a$$

Then, the packed version of a program $\mathbf{p}$ has a structure similar to the one presented in Figure 1, where the value b is the result of the computation $\mathbf{encode}(\mathbf{p})$.

$$x := b;$$
$$x := \mathbf{decode}(x);$$
$$\mathbf{exec}(x);$$

Figure 1. A simple packed program

We observe that from a compilation-theoretic point of view:

- the first line corresponds to a *specialization pattern*,
- the second and third lines correspond to an interpreter for encoded programs on the target architecture.

In Figure 2, we present such an interpreter written in assembly language where the programs are decoded with a simple `xor` by constant value `0xdeadbabe`. The pointer to the payload is noted `x` and `len(x)` is the length of the payload.

2.3 Self-modifying code elimination

As said above, in our framework, the protection of packed programs consists in a factitious compilation process, which yields a program with a decoding overhead. Therefore, we can see unpacking as a process aiming at removing this overhead.

```
  mov ecx, (len(x)/4) ; init decryption
  lea eax, x

loop:                 ; decryption loop
  mov edx, [eax]
  xor edx, 0xdeadbabe
  mov [eax], edx
  add eax, 0x4
  dec ecx
  jnz loop            ; end of loop

  call x              ; call payload
```

Figure 2. Interpreter for xored programs

From a compilation perspective, this corresponds to an optimization process. We then propose to simplify the packed program by restoring the dynamic code observed during a run and suppressing the instructions that generated it.

This idea appears clearly in the program of Figure 1, where the propagation of the constant value b would lead to the following program:

$$x := \mathbf{decode}(b);$$
$$\mathbf{exec}(x);$$

Then replacing the expression $\mathbf{decode}(b)$ by $\mathbf{p}$ and propagating this value would lead to the following program:

$$\mathbf{exec}(\mathbf{p});$$

Finally, the interpretation becomes useless and a trivial code alignment would lead to the program $\mathbf{p}$. We conclude that the original program can be recovered by using classic optimization techniques, namely:

- constant propagation,
- code alignment.

Unfortunately, it is highly difficult to statically optimize programs, particularly for binary code. However, at runtime, more information on the execution context is available and it is easier to optimize specific code regions. This is the premise

of just-in-time compilers which rely on runtime information to specialize code regions and boost performance, as described in [18]. From this observation, we propose to apply our approach to the easier context of execution traces.

It will therefore consist in executing the subject program, then using the runtime information to inline the hidden code and finally simplifying the execution trace to make it free of self-modifications.

3 Execution trace flattening

3.1 Overview of the algorithm

We now present the algorithm used to transform a self-modifying program into a flat execution trace. This process is achieved in three steps:

- extracting an execution trace of the packed program,
- backtracking dependencies of self-modified code,
- simplifying the execution trace.

Execution trace. An execution trace is a sequence of system states resulting from the execution of the subject program. In order to extract such a trace, we have to log changes to the execution context (register and memory values) each time an instruction is executed.

Dataflow backtracking. We are specifically interested in memory regions that are written and then executed, because they indicate the presence of dynamic code. We mark such memory addresses and we backtrack the dataflow dependencies through a slicing algorithm to find the instructions that affected the marked memory region. In other terms, we backtrack the unpacking stub from the payload. We do this by computing "def-use" chains: there is a def-use dependency between two instructions i_1 and i_2 if i_2 reads data written by i_1.

Trace optimization. The next step is to optimize the trace by deleting all instructions that spawn dynamic code. For this, we simply remove the code backtracked in the first step. This process uses a well-known dynamic slicing algorithm with a deletion constraint on dynamic code. We refer to [23] for further theoretical details on it.

Program recovery. Sometimes it is possible to recompile the optimized trace into an executable. For this we fold the trace into a program using the observed addresses. This functionality is still unstable in our current implementation. The main limitation is due to the possibility for a packer to yield different code blocks at the same address. At the current state of prototyping, this is not a problem if the generated code is position independent: we pad the addresses with the layer number and we recover the control flow by adding offsets to jump targets. Dealing with position dependent code is more difficult and requires a thorough data dependency analysis. We are currently working on this aspect.

3.2 Implementation details

We make use of several tools to achieve flattening. First, we extract the execution trace by running the program within a monitored environment with dynamic instrumentation capabilities. This process uses of a specialized sandbox which monitors and analyzes data dependencies on code access. We rely on TEMU [20] (a whole-system emulator for binary analysis based on QEMU) for program tracing, but we could also use other dynamic analysis systems such as Bochs [1], DynamoRIO [3] or Pin [13].

To express data dependencies, we model the instruction semantics in a Pascal-like language. This dependency framework has been developed as an extra analysis layer over Vine [20].

Next, the backtracking of data dependencies is achieved by a slicing algorithm built from the survey of several dynamic slicing algorithms described in [23]. The optimization of the trace is straightforward: it consists in the deletion of the backtracked code.

3.3 Proof of concept

We now illustrate the flattening procedure step by step on a self-modifying program example. It makes use of the interpreter for xored programs of Figure 2 where `x` is set to a xored code that computes the factorial of 4. The code of the resulting program in presented in Figure 3.

```
payload:                ; xored code of 4!
  .dd "\x07\xbe\xad\xde\xbe\x02\xac\xde"
  .dd "\xbe\xba\x5a\x3f\xf7\xcf\x56\x1d"

  mov ecx, 4            ; init decryption
  lea eax, payload

loop:                   ; decryption loop
  mov edx, [eax]
  xor edx, 0xdeadbabe
  mov [eax], edx
  add eax, 0x4
  dec ecx
  jnz loop              ; end of loop

  call payload          ; call payload
```

Figure 3. Self-modifying program

First, let us follow the execution of this program.

1. The first instructions are the initial setup for the decoding loop. Register `ecx` is initialized with the length (i.e. number of 32-bit values) of the encoded buffer, and `eax` points to the beginning of the buffer.

2. Then, the decryption loop performs the xor operation with constant 0xdeadbabe on each 32-bit value of the buffer.

3. Finally, the control is transferred to the now decrypted buffer.

The execution of this program produces the trace presented in Table 1. In this trace, we observe that the code region 0x12ff64–0x12ff73 is written and executed. Then, the slicing algorithm backtracks the instructions that are highlighted with a bullet in Table 1.

Time	Address	Instruction	
1	0x401048	`mov ecx, 0x4`	
2	0x40104b	`lea eax, 0x12ff64`	•
3,9,15,21	0x40104e	`mov edx, [eax]`	•
4,10,16,22	0x401050	`xor edx, 0xdeadbabe`	•
5,11,17,23	0x401056	`mov [eax], edx`	•
6,12,18,24	0x401058	`add eax, 0x4`	•
7,13,19,25	0x40105b	`dec ecx`	
8,14,20,26	0x40105c	`jne 0x40104e`	
27	0x40105e	`call 0x12ff64`	
28	0x12ff64	`mov ecx, 0x4`	
29	0x12ff69	`mov eax, 0x1`	
30,33,36,39	0x12ff6e	`mul ecx`	
31,34,37,40	0x12ff70	`dec ecx`	
32,35,38,41	0x12ff71	`jne 0x12ff6e`	
42	0x12ff73	`ret`	

Table 1. Execution trace

The backtracking procedure works as follows. The algorithm linearly processes the execution to identify memory regions that have been written and then executed. In Table 1 we observe write accesses at times 5, 7, 11 and 23, the corresponding accessed region is then executed. From this, we backtrack the dataflow dependencies on this region, making use of the propagation relations given by Vine [20].

For instance, the data used in instruction `mov [eax], edx` at time 17 depends on the instruction `xor edx, 0xdeadbabe` at time 16.

After this process, we simply `nop` the backtracked instructions. We obtain the execution trace presented in Table 2. It is free of self-modification and it clearly computes the factorial of 4. We observe that the instruction `add eax, 0x4` at time 24 has not be deleted, this is because it does not depend on any subsequent write access to an executed region. The resulting trace includes a superfluous loop which is subject to further optimization focused on *dead code elimination*. This refinement work flow is a common practice in program optimization.

Time	Address	Instruction
1	0x401048	`mov ecx, 0x4`
2	0x40104b	`nop`
3,9,15,21	0x40104e	`nop`
4,10,16,22	0x401050	`nop`
5,11,17,23	0x401056	`nop`
6,12,18	0x401058	`nop`
24	0x401058	`add eax, 0x4`
7,13,19,25	0x40105b	`dec ecx`
8,14,20,26	0x40105c	`jne 0x40104e`
27	0x40105e	`call 0x12ff64`
28	0x12ff64	`mov ecx, 0x4`
29	0x12ff69	`mov eax, 0x1`
30,33,36,39	0x12ff6e	`mul ecx`
31,34,37,40	0x12ff70	`dec ecx`
32,35,38,41	0x12ff71	`jne 0x12ff6e`
42	0x12ff73	`ret`

Table 2. Flattened execution trace

4 Scope and limitations

In this paper, we proposed a generic technique to identify dependencies between dynamic code and the code that generates it at runtime. In particular, if this operation is unconditional (i.e. it does not depend on the program input), the hidden code can be restored by observing a single execution path. We assert that most packers work by unconditionally restoring the original program, therefore our technique should eventually give good results on packed samples. Preliminary testing on x86 self-modifying code confirms this intuition. However, we are naturally confronted to a number of limitations.

Conditional behaviors. The trace on which we perform the optimizations depends on a specific input. Therefore, any generalization based on the transformed trace can be misleading if the generated code depends on the input.

Multiple layered unpacking. In the case study, we made the assumption that the code was unpacked in a single layer. In essence, the same optimization technique could be applied recursively to other code layers, though this has not been tested yet.

External code. The previous case study focuses on user-mode x86 with no library or system calls, which would have to be handled specially.

Control dependencies. As usual with unstructured control-flow, we only focus on data dependencies. Control dependencies would require some degree of static analysis [17, 5, 24, 25], which is undecidable on binary programs in the general case.

Transparency of the transformation. We can not ensure that the output binary works exactly like the original one, since it has been partly rewritten. For instance, in addition to perform self-modifications, the unpacking stub may also try to detect if it has been tampered with. In this case, we would remove the self-modifications but not the integrity checking, which would fail. However, the resulting binary would be analyzable statically and could therefore be helpful.

5 Further work

This study is the first step toward an ambitious research program. We are currently working in two main directions to enhance trace flattening and to tackle the current limitations of our implementation.

First, we have underlined a strong relation between unpacking and program optimization. We want to push this intuition forward by using standard optimization frameworks. The difficulty is that we have to translate assembly code into a higher level language. As a result we are confronted to the complex semantics of the x86 assembly language. To overcome this issue we are working on the formal semantics specification given in [16].

Second, we are designing a new version of the sandbox to force execution contexts in order to increase code coverage. For this we have inspired from [15] where the authors propose a dynamic analysis framework to explore as much execution

paths as possible. Within the context of program flattening, the idea would be to generate several execution traces to cover the whole control flow graph of the subject binary. Using the same process as before we would be able to backtrack self-modifying code. Then, we could recover a flat executable if the different execution traces are consistent.

References

[1] Bochs. http://bochs.sourceforge.net/.

[2] L. Böhne. Pandora's bochs: Automatic unpacking of malware. *University of Mannheim*, 2008.

[3] D.L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[4] S. Cesare and Y. Xiang. A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 721–728. IEEE, 2010.

[5] 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*, page 206. ACM, 2007.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[7] P. Ferrie. Anti-unpacker tricks. In *Proc. of the 2nd International CARO Workshop*, 2008.

[8] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[9] F. Guo, P. Ferrie, and T. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[11] S. Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3(3):221–236, 2007.

[12] M.G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, page 53. ACM, 2007.

[13] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[14] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 431–441. Citeseer, 2007.

[15] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy, 2007. SP'07*, pages 231–245, 2007.

[16] M.O. Myreen. Verified just-in-time compiler on x86. *ACM SIGPLAN Notices*, 45(1):107–118, 2010.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis,

and signature generation of exploits on commodity software. In *In Proceedings of the Network and Distributed System Security Symposium*, 2005.

[18] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[19] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300, 2006.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. *Information Systems Security*, pages 1–25, 2008.

[21] L. Sun, T. Ebringer, and S. Boztas. Hump-and-dump: efficient generic unpacking using an ordered address execution histogram. *Department of Computer Science and Software Engineering, The University of Melbourne, Australia*, 2008.

[22] P. Szor. *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

[23] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[24] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[25] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, page 127. ACM, 2007.