Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming

Dennis Andriesse, Herbert Bos and Asia Slowinska
System and Network Security Group
VU University Amsterdam
{da.andriesse,herbertb,asia}@few.vu.nl

***Abstract*—*Parallax* is a novel self-contained code integrity verification approach, that protects instructions by overlapping Return-Oriented Programming (ROP) gadgets with them. Our technique implicitly verifies integrity by translating selected code (*verification code*) into ROP code which uses gadgets scattered over the binary. Tampering with the protected instructions destroys the gadgets they contain, so that the verification code fails, thereby preventing the adversary from using the modified binary. Unlike prior solutions, *Parallax* does not rely on code checksumming, so it is not vulnerable to instruction cache modification attacks which affect checksumming techniques. Further, unlike previous algorithms which withstand such attacks, *Parallax* does not compute hashes of the execution state, and can thus protect code with non-deterministic state. *Parallax* limits performance overhead to the verification code, while the protected code executes at its normal speed. This allows us to protect performance-critical code, and confine the slowdown to other code regions. Our experiments show that *Parallax* can protect up to 90% of code bytes, including most control flow instructions, with a performance overhead of under 4%.**

I. Introduction

Code integrity verification is an anti-tampering primitive which aims to ensure that code executes as intended on a hostile host [36], without being modified by an adversary. Self-verification is a subclass of code verification, which provides integrity checking without requiring specialized hardware, such as trusted execution modules, and without the use of remote verification servers.

Code protection primitives like integrity verification are widely used in practice to delay reverse engineering attacks, and to deter non-persistent adversaries. Code protection is commonly used by malware to prolong its lifespan and monetization period [7, 29, 30], but it is also used to protect benign programs against software cracking [6]. Furthermore, code integrity verification in particular can also defend against certain parasitic malware techniques, which inject inline hooks or code into processes [37] or executables [34].

Preventing malicious code injection is not only crucial to the security of end-user computer systems, but also as a defense against high-profile attacks, the importance of which is witnessed by recent targeted threats like Stuxnet [17] and Gauss [21]. Recently, the American Institute of Aeronautics and Astronautics launched a code protection initiative to prevent attacks against aviation control systems [3]. The United States Department of Defense has also expressed interest in code protection for use in hardened computing centers, as well as real-time software used in weapon systems which may fall into enemy hands [35].

Most existing code self-verification algorithms work by computing checksums over protected code regions, and verifying that these checksums are as expected. Using several cross-verifying checksummed code regions, such algorithms can provide fairly strong tamperproofing. Unfortunately, Wurster et al. have shown that all such algorithms are inherently vulnerable to automated attacks which exploit the distinct handling of code and data in modern processors [36]. Wurster et al. implement a kernel patch which allows attackers to freely tamper with the code in the processor's instruction cache, while leaving the data cache entirely untouched. This completely circumvents checksumming algorithms, as these treat code as data, thus fetching it from the data cache instead of the instruction cache.

The foremost algorithm designed to defeat this attack is oblivious hashing (OH) [13, 20]. Instead of directly checking code integrity, oblivious hashing intersperses hashing instructions with the protected code, which build runtime hashes of the execution state. The integrity is then verified by checking that the computed hashes correspond to known correct values. However, this technique can only verify deterministic execution state, of which the expected hash is known at compile time. This means that OH cannot protect code which involves non-deterministic inputs, such as environment parameters or user input. Additionally, the hashes used to verify the state are found using dynamic testing, limiting the protection to code paths exercised in these tests.

We propose a novel code self-verification approach, which is based on Return-Oriented Programming (ROP). ROP was originally proposed as an exploitation technique which allows arbitrary code execution in the presence of W$\oplus$X [33]. ROP uses short return-terminated instruction sequences, called *gadgets*, which are chained together by arranging their addresses on the stack such that each terminating return causes a control transfer to the next gadget. If a sufficient set of gadgets is available, ROP is a Turing-complete programming technique which can implement arbitrary computations on top of a host program. A Turing-complete gadget set exists in most programs [32]. We refer to an arrangement of gadget addresses into a ROP program as a *ROP chain*.

Our code verification approach protects code by overlapping ROP gadgets with it. Then, selected instructions from the protected program are translated into ROP chains which use the overlapping gadgets. Since tampering with the gadgets causes the translated instructions to malfunction, this implicitly verifies the integrity of the protected code. Thus, we refer to these translated instructions as *verification code*. Our notion of verification code can be seen as a generalization of code hiding techniques based on function reuse [23]. We show in Section VI that the verification code is itself also tamper resistant.

Our technique has several advantages over prior work.

1) The code verification does not use checksumming, and is thus immune to the attack of Wurster et al.
2) In contrast to oblivious hashing, no prior knowledge of the runtime state is required. Therefore, our technique can protect non-deterministic code regions. Furthermore, we apply this protection statically, so it is oblivious to dynamic code coverage limitations.
3) The overlapping gadgets do not slow down code they protect. Instead, performance overhead is confined to the verification code using the gadgets. This makes it possible to tamperproof performance-critical code while confining the performance degradation elsewhere. In contrast, oblivious hashing slows down protected code by interspersing hashing instructions with it.
4) We show in Section VII that our technique can protect up to 90% of code bytes at a performance overhead of less than 4%. As argued in Section VIII, non-deterministic control flow decisions are among the most likely attack targets. Thus, our protectability rate exceeds that of oblivious hashing [13], and we protect crucial instructions which OH cannot.
5) In contrast to prior work, including oblivious hashing, our approach lends itself to binary-level implementation, and does not inherently require source. This enables the protection of legacy binaries.

Since the verification code uses ROP, it requires a set of gadgets overlapping with the instructions we protect. We both use gadgets already present in the host binary, and statically rewrite the binary to craft new ones. Since a Turing-complete gadget set is already present in most programs [32], the additional tamperproofing gadgets generally do not increase the vulnerability of protected programs to ROP attacks.

As a proof of concept, we built a prototype implementation of our technique for the x86 platform, called *Parallax*. It uses binary rewriting to create protective gadgets, and builds on ROP compiler functionality to generate verification code. Our proof of concept uses source to simplify binary rewriting, and also offers the option of selecting verification code at the source level. However, this is not a requirement of our technique, which can be implemented entirely at the binary level.

The rest of this paper is organized as follows. Section II discusses background and threat assumptions, while Section III provides an overview of *Parallax*. In Section IV, we describe the rules we use to craft protective gadgets. Section V discusses the implementation of verification code, and Section VI discusses the attack resistance of *Parallax*. Evaluation results and limitations of *Parallax* are discussed in Sections VII and VIII. We discuss related work and summarize our conclusions in Sections IX and X.

II. Background

This section describes the workings of Return-Oriented Programming, upon which we base our technique. Furthermore, we describe the threat model which *Parallax* assumes.

A. Return-Oriented Programming

ROP was originally proposed in 2007 as an exploitation technique designed to circumvent memory protection mechanisms like W⊕X [33]. ROP makes use of short instruction sequences found in a host program's memory space, called *gadgets*, each of which ends in a return instruction. Each gadget typically performs a basic operation, such as addition or logical comparison. Gadgets can be part of the host program's normal instructions, but can also be unaligned instruction sequences embedded in the normal instruction stream. A ROP program consists of a chain of gadget addresses on the stack, such that the return instruction terminating each gadget transfers control to the next gadget in the chain.

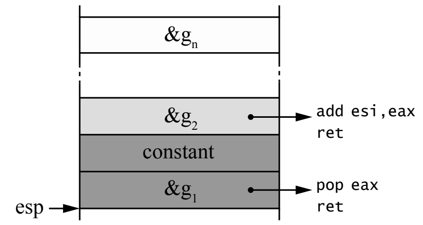


Fig. 1: An example ROP chain. Gadget g_1 loads a constant into `eax`, which is then added to `esi` by g_2.

Figure 1 illustrates an example ROP chain. Initially, the stack pointer (`esp`) points to the address of the first gadget g_1 in the chain. Upon execution of a return instruction, control is transferred to this gadget. It performs a `pop` instruction, which loads a constant arranged on the stack into the `eax` register, and increments `esp` to point to gadget g_2. Then, the `ret` instruction of gadget g_1 transfers control to gadget g_2, which adds the constant in `eax` to the `esi` register. Gadget g_2 then returns to gadget g_3, and so on, until all gadgets $g_1, \dots, g_n$ have been executed.

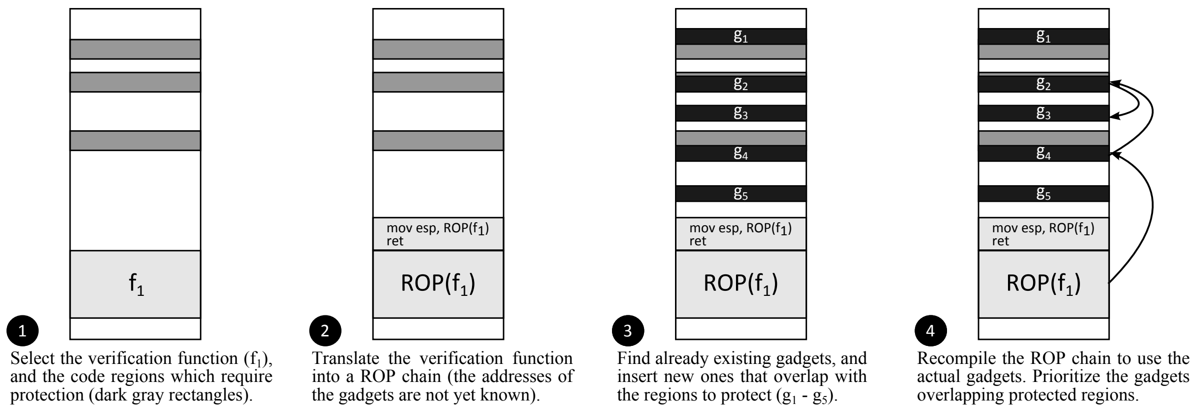
B. Threat Model

Parallax assumes the *hostile host* threat model [36], which is the standard model for tamperproofing techniques. It assumes that the tamperproofed application is executed on a system controlled by a hostile user, which has full control over the runtime environment, and may arbitrarily modify the tamperproofed executable itself. This includes alterations made during runtime debugging, as well as static code patching. The intent of the hostile user is typically to circumvent access controls in the protected application, such as anti-debugging checks or license verifications. The challenge for our tamperproofing technique is thus to maximize the effort required by the hostile user to successfully tamper with the protected application, without assuming any trusted components in the runtime environment.

III. Overview

In this section, we give an overview of how *Parallax* implements its protection mechanism. Our technique protects against both static and dynamic code modification. Thus, we protect against attacks ranging from the circumvention of anti-debugging checks, to large-scale software cracking. Figure 2 illustrates how *Parallax* protects a binary.

To protect a binary, we select one or more code fragments at the source or binary level for use as verification code (step ① in Figure 2). In Section VII-B, we describe our strategy to do this automatically. Additionally, we determine a list of



f$_1$
mov esp, ROP(f$_1$)
ret
ROP(f$_1$)
g$_1$
g$_2$
g$_3$
g$_4$
g$_5$
1 Select the verification function (f$_1$), and the code regions which require protection (dark gray rectangles).
2 Translate the verification function into a ROP chain (the addresses of the gadgets are not yet known).
3 Find already existing gadgets, and insert new ones that overlap with the regions to protect (g$_1$ - g$_5$).
4 Recompile the ROP chain to use the actual gadgets. Prioritize the gadgets overlapping protected regions.

Fig. 2: A high-level overview of *Parallax*.

instructions to protect. If source is available, these are selected at the statement or function level, and then mapped to the binary level after compilation using debugging symbols. If only a binary is available, protection is assigned at the instruction or function level.

Parallax begins by translating the selected verification code into one or more ROP chains (sequences of ROP gadgets) ②. These chains use placeholder gadget addresses, since the final addresses are not yet known at this point. Eventually, these placeholders will be replaced by gadget addresses in the protected code, so that executing the verification code implicitly verifies that the protected code is still intact. Along with the ROP chains, *Parallax* inserts a loader routine to bootstrap them. Optionally, the binary to protect is compiled from source if not operating on a legacy binary.

Next, *Parallax* creates a collection of all gadgets available in the binary ③. First, any existing gadgets are added to the collection. Then, *Parallax* walks through the list of instructions which were selected for protection. For every such instruction, *Parallax* examines if it can be augmented with an overlapping gadget. If so, it inserts a gadget using binary rewriting, and adds this new gadget to the collection. The gadgets are denoted as $g_1, \ldots, g_5$ in Figure 2. Our strategy for crafting overlapping gadgets is discussed in Section IV.

Note that we do not require the inserted overlapping gadgets to form a Turing-complete set, since most binaries contain a Turing-complete gadget set by default [32]. If not, a standard set of non-overlapping gadgets can be inserted into the binary to augment the protective gadgets already inserted.

Finally, *Parallax* creates a *gadget mapping* which categorizes the available gadgets in the binary into a set of types; for instance, memory stores and register moves. The gadget mapping is then used to recompile the verification code such that it uses actual gadgets instead of placeholder addresses ④. During compilation of the verification code, overlapping gadgets are always preferred over non-overlapping gadgets. Tampering with the protected instructions modifies the code bytes of the overlapping gadgets, thereby invalidating them. Such changes are implicitly detected by the verification code, which malfunctions if the integrity of the gadgets it uses is violated. We discuss the tampering and analysis resistance of verification code in Sections V and VI.

Listing 2 An attempt to disable the ptrace detector.

```
(gdb) set *(unsigned char*)0x08048479=0x90
(gdb) set *(unsigned char*)0x0804847a=0x90
```

IV. Protecting Code Integrity

This section discusses the creation of ROP gadgets which overlap with existing code, and protect the code integrity. As discussed in Section III, these gadgets need not form a Turing-complete set. Instead, the focus is on gadgets which have maximal overlap with the protected instructions. The creation of verification code which uses the gadgets is discussed in Section V. We provide an example of gadget insertion in Section IV-A, and generalize it in Section IV-B by describing the rules which *Parallax* uses to craft overlapping gadgets.

A. A Tamperproofed Ptrace Detector

We provide a running example of a ptrace detection function augmented with overlapping gadgets. We compiled this function with gcc 4.6.3, and then used *Parallax* to search for locations where overlapping gadgets could be inserted. In the example, we manually chose which instructions to protect from the list of possible locations emitted by *Parallax*. To avoid manual effort, it is also possible to input a list of functions to protect, and rely on *Parallax* to overlap gadgets with as many instructions in these functions as possible. An alternative approach, if source is available, is to select high-level source lines to protect, and use debugging symbols to map these onto the associated machine instructions. The rules *Parallax* uses to create gadgets are discussed in Section IV-B.

Listing 1 shows a disassembly dump of our tamperproofed ptrace detector. For clarity, we have shortened addresses such as `08048438` to `n+38`. We first describe the purpose of the ptrace detector, and then elaborate on how it is protected.

The ptrace detector checks if a process is being debugged using ptrace. To achieve this, the detector calls the `ptrace` system call, requesting a trace of the host process. If a debugger is already attached, this call fails, and the debugger is thus detected. In the example, the detector jumps to a `cleanup_and_exit` function if a debugger is detected.

Attackers commonly attempt to circumvent such anti-debugging code by modifying it at runtime, as shown in

Listing 1 A ptrace detector with gadgets (shaded) overlapping sensitive areas.

```
n+38 <cleanup_and_exit>:                                            n+32 <cleanup_and_exit>:
n+38: 55                         push ebp           relocate   -->  n+32: 55                         push ebp
n+39: 89 e5                      mov  ebp,esp                       n+33: 89 e5                      mov  ebp,esp
n+3b: 83 ec 18                   sub  esp,24                        n+35: 83 ec 18                   sub  esp,24
n+3e: 89 04 24                   mov  [esp],eax                     n+38: 89 04 24                   mov  [esp],eax
n+41: e8 d5 fe ff ff             call exit@plt                      n+3b: e8 d5 fe ff ff             call exit@plt

n+46 <check_ptrace>:                                                n+46 <check_ptrace>:
n+46: 55                         push ebp                           n+46: 55                         push ebp
n+47: 89 e5                      mov  ebp,esp                       n+47: 89 e5                      mov  ebp,esp
n+49: 83 ec 18                   sub  esp,24                        n+49: 83 ec 18                   sub  esp,24
n+4c: c7 44 24 0c 00 00 00 00    mov  [esp+0xc],0                   n+4c: c7 44 24 0c 00 00 00 00    mov  [esp+0xc],0
n+54: c7 44 24 08 00 00 00 00    mov  [esp+0x8],0                   n+54: c7 44 24 08 00 00 00 00    mov  [esp+0x8],0
n+5c: c7 44 24 04 00 00 00 00    mov  [esp+0x4],0                   n+5c: c7 44 24 04 00 00 00 00    mov  [esp+0x4],0
n+64: c7 04 24 00 00 00 00       mov  [esp],0                       n+64: c7 04 24 00 00 00 00       mov  [esp],0
                                                existing far ret
n+6b: e8 cb fe ff ff             call ptrace@plt               -->  n+6b: e8 cb fe ff ff             call ptrace@plt
n+70: 85 c0                      test eax,eax                       n+70: 85 c0                      test eax,eax
n+72: 79 07                      jns  n+7b                          n+72: 79 07                      jns  n+7b
                                                modify exit arg
n+74: b8 01 00 00 00             mov  eax,1                    -->  n+74: b8 c3 00 00 00             mov  eax,0xc3
                                                modify target
n+79: eb bd                      jmp  n+38                     -->  n+79: eb c3                      jmp  n+32
n+7b: b8 00 00 00 00             mov  eax,0                         n+7b: b8 00 00 00 00             mov  eax,0
n+80: c9                         leave                              n+80: c9                         leave
n+81: c3                         ret                                n+81: c3                         ret
```

Listing 2. There, an adversary overwrites the jump to the `cleanup_and_exit` function at address `n+79` with `nop` instructions. The goal of this attack is to redirect control to a successful return even if a debugger is attached.

Overlapping gadgets defend against this attack class, as they are destroyed if the code they overlap with is modified. As mentioned in Section III, this is detected when the verification code using the gadgets fails to execute. Note that an adversary could also modify the call to `check_ptrace` itself. As we show in Section VII-A, *Parallax* can protect up to 90% of the binary, allowing us to defend against such attacks by inserting protective gadgets beyond the primary list of instructions to protect. While this example focuses on runtime code modification, *Parallax* also prevents static code patching, used in software cracking.

In Listing 1, four key code areas which adversaries are likely to target have been protected using three overlapping gadgets. The first two locations are (1) the call to `ptrace` itself, at address `n+6b`, and (2) the first argument to `ptrace`, at address `n+64`, which requests a trace of the host process. An adversary may eliminate the call, so that execution always falls through to the successful return code at the end of the function. Also, an adversary may modify the call argument to request another action from `ptrace` instead of a trace of the host process. Both the call and its first argument are protected by a seven byte long overlapping gadget starting at address `n+66`. This is an already existing gadget, which *Parallax* found without making any code modifications. The gadget consists of the instructions `and al,0; add [eax],al; add al,ch; retf`, and can be used to move the contents of the `ch` register into the `al` register (the memory write can be ignored, since `al` is zeroed out).

Note that it is also possible to protect the remaining `ptrace` arguments at addresses `n+4c` through `n+5c`. One possible way to protect these is to use the immediate splitting rule, discussed in Section IV-B2. For simplicity of the example, we do not show these modifications in Listing 1. However, we provide a separate example of the immediate splitting rule in Section IV-B2.

The third location which may be attacked is (3) the jump to the `cleanup_and_exit` function, at address `n+79`, which is taken if a debugger is detected. Eliminating this jump would again cause control to fall through to the successful return at the end of the function, even if the call to `ptrace` failed. *Parallax* protects this jump by relocating the `cleanup_and_exit` function, and modifying the jump offset to encode the `ret` instruction for a gadget. The gadget starts at address `n+78`, and contains instructions `add bl,ch; ret`.

Finally, the anti-debugging code could be disabled by (4) rewriting the `jns` instruction at address `n+72` to an unconditional `jmp` instruction, so that the code always jumps to a successful return. *Parallax* identifies two possible ways to protect against this. The first is to modify the immediate operand of the `mov` instruction at address `n+74`, such that its least significant byte encodes a `ret` instruction. This creates a five byte long gadget at address `n+71`, consisting of the instructions `sar byte [ecx+0x7],0xb8; ret`. This gadget fills the memory byte at address `[ecx+0x7]` with the sign of the byte it contains (the bits are either all set to 0, or all set to 1). The `mov` operand is an exit status, and can be safely modified assuming that the exit semantics differentiate only between zero and non-zero (see Section IV-B).

An alternative way to protect the `jns` is to inject a spurious instruction directly after it, which encodes the missing part of a partial gadget. In the example, we did not use spurious instructions, to show that no added code is needed to protect the function.

B. Binary Rewriting Rules

This section describes the binary rewriting rules *Parallax* uses to augment instructions with overlapping gadgets. The added gadgets do not induce any performance overhead on the

Listing 3 A split `mov` with overlapping gadgets (shaded).

```
mov eax,1
```

(a) Original code.

```
b8 01 01 c3 00 mov eax,0xc30101
35 00 01 c3 00 xor eax,0xc30100
```

(b) Protected code.

protected code, except where explicitly noted. In Section VII-A, we measure the coverage of each of these rules. We use binary rewriting techniques for legacy binaries explored in prior work [22, 38].

1) Existing gadgets: Parallax searches for any existing gadgets which can be used to protect code integrity. The use of existing gadgets is advantageous, as it requires no modifications to the protected code regions. In Section VII-A, we find that 3%–6% of the code bytes in our test cases is protectable using existing gadgets.

2) Modified immediate operands: One rule used by *Parallax* to create new gadgets is that a partial gadget may be combined with an adjacent immediate operand if this operand can be modified to encode the missing portion of the desired gadget. In Listing 1, this rule has been applied in the operand of the instruction at address `n+74`. We distinguish two ways in which immediate operands can be safely modified.

First, depending on the instruction type, immediates can be modified by splitting up their parent instructions. For instance, an addition can be split into two additions or subtractions, where the first takes an arbitrary operand, and the second compensates as required. Similarly, immediate operands of `mov` instructions can be modified to encode a gadget, and this modification can then be compensated for using bitwise operations on the destination operand. As an example of this rule, Listing 3 shows how the immediate operand of a `mov` instruction is modified and combined with an `xor` instruction to compensate for the modifications.

Instruction splitting induces a small performance overhead on the protected code. Additionally, it may require the insertion of code to save and restore the CPU status register.

Second, it is generally possible to freely modify immediates which set `eax` before a return, or push the status of the `exit` function. This is because return value and exit status semantics commonly distinguish only between zero and non-zero. This rule can be disabled for conflicting semantics.

3) Rearranged code and data: Parallax also attempts to encode missing parts of gadgets in addresses and jump offsets by strategically aligning functions and global variables. For instance, in the example shown in Listing 1, we have forced the creation of a `ret` instruction by aligning the `cleanup_and_exit` function such that the jump offset at address `n+79` is equal to `0xc3` (the `ret` opcode).

4) Spurious instructions: Spurious instructions which contain (parts of) gadgets can be inserted at any place in the code, as long as care is taken to ensure that their side-effects do not influence the semantics of the original code. This can be ensured by saving and restoring the program state at each location where spurious instructions are inserted. Alternatively, side-effect analysis can be performed on the inserted instructions using frameworks such as BAP [9].

The main benefit of spurious instructions is that they can always be inserted to encode missing parts of gadgets. However, because spurious instructions induce a slowdown on the protected code, it is best to avoid them if possible.

5) Far-return gadgets: Far returns (`retf`) are quite rare in compiler-generated x86 code. Nevertheless, gadgets ending in far returns can sometimes be used to protect code bytes, as was done at address `n+66` in Listing 1. *Parallax* searches for existing far-return gadgets in the same way as for near-return gadgets.

6) Using add for memory operations: One of the most useful instruction families for gadgets is the `add` family. This is because the opcodes of `add` range from `0x00` to `0x05`, which are very common values in immediate operands. Listing 1 contains several gadgets which use `add` instructions, such as the gadget protecting the call to `ptrace`.

Next to implementing additions, `add` instructions with memory operands can also be used as loads and stores. For instance, `add [ecx],eax` is a store of the value in `eax` to the address in `ecx`, if this memory is initially zero.

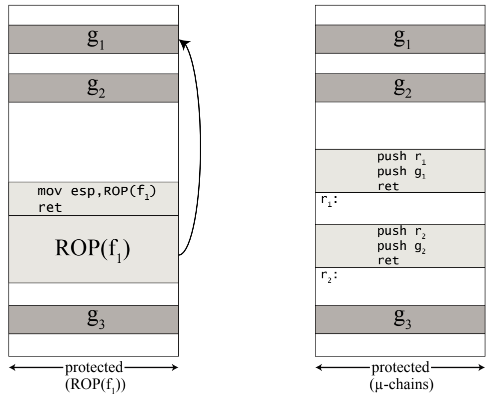
V. Verifying Code Integrity

In Section IV, we discussed the creation of overlapping gadgets for code protection. To protect their parent instructions, the integrity of these gadgets must be verified by one or more ROP chains. In this section, we discuss the translation of existing (source or binary) code from the protected program into ROP chains which act as verification code. These ROP chains use the gadgets contained in the protected code regions. We stress that the verification code does not perform any active verification, checksumming or otherwise. Instead, it detects and responds to tampering in a completely passive way, by malfunctioning if the gadgets in protected code regions are damaged. We implement verification code at function granularity, meaning that whole functions from the original program are translated to ROP code. For brevity, we refer to a function-level verification ROP chain as a *function chain.* In Section V-C, we also briefly report on our experiences with instruction-level verification.

A. Implementation of Function Chains

Function chains were already briefly discussed in Section III. Figure 3a illustrates a binary protected using function chains. As discussed in Section III, the protected binary contains several gadgets g_1, g_2, g_3, which are crafted such that they overlap with instructions which must be protected. Furthermore, a selected function f_1 from the protected binary's code section is translated into equivalent ROP verification code, denoted as $ROP(f_1)$. Additionally, a small amount of loader code is inserted, which is responsible for starting the execution of the verification code. The minimum operations required for this are (1) pointing the stack pointer to the beginning of $ROP(f_1)$, and (2) executing a return instruction to transfer control to the first gadget in the verification code.

In our *Parallax* prototype, we implemented function-level verification on top of a modified version of the open source ROP compiler ROPC [2], which is based on Q [32]. Our prototype loader code, which bootstraps the execution of the function chains, is slightly more extensive than shown in Figure 3a. Particularly, in addition to pointing the stack pointer to the start



(a) A function chain. (b) Scattered μ-chains.

Fig. 3: Verification at function and instruction level.

of a function chain and executing a return, we also ensure that execution continues cleanly after the function chain is complete. To achieve this, the loader code appends an epilogue to each function chain before transferring control to it. The epilogue consists of the address of a `pop esp` gadget, followed by a stack address in the original stack frame of the calling function. At this stack address, the return address for the function chain is stored. Before the epilogue's `pop esp` gadget is executed, the stack pointer points inside the function chain. The `pop esp` points it back into the calling function frame, to the stack location containing the function chain's return address. Now, when the function chain returns, this transfers control back to the calling function, and program execution continues normally.

In addition to the epilogue, we perform a `pushad` directly before, and a `popad` directly after each function chain. These instructions save and restore the register state, preventing problems due to registers clobbered by the function chain.

B. *Dynamically Generated Function Chains*

Function chains can reside in data memory, which is writable even with W$\oplus$X protection on. This means that it is possible to generate function chains at runtime. *Parallax* implements optional support for this. Dynamic function chain generation has several advantages. (1) It allows for encrypted and self-modifying function chains, which are more resistant to analysis than their non-dynamic counterparts. We evaluate the performance of RC4-encrypted and xor-encrypted function chains in Section VII-B. (2) Multiple instances of the same function chain can be generated probabilistically, with each instance using a different set of semantically equivalent gadgets. This allows a small function chain to verify a large set of gadgets, checking a subset each time it is executed. The tradeoff of this approach is that the protection of each gadget becomes probabilistic, rather than deterministic.

Specifically, let $T := \{t_1, \dots, t_n\}$ be the set of used gadget types in the function chain, and for $1 \le i \le n$, define $G_i := \{g \mid g \text{ implements } t_i\}$. Thus, each G_i is the set of all gadgets which implement gadget type t_i. For probabilistically generated function chains, we use an extended notion of the gadget types mentioned in Section III, which defines not only the operation implemented by a gadget, but also its operand registers and memory locations. Then, for every operation, the function

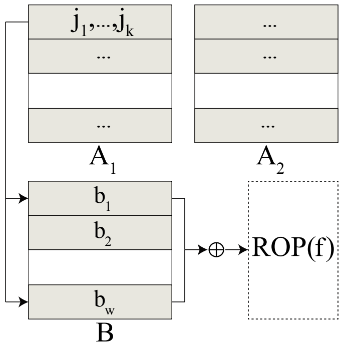


Fig. 4: Generating a function chain by combining vectors from a basis B, indexed by arrays A_1 and A_2.

chain can probabilistically choose a gadget $g \in G_i$. In total, this yields $\prod_{i=1}^{n} |G_i|$ possible distinct gadget subsets which can be checked by the same function chain. Because the used subset of gadgets is probabilistic, it is hard for an adversary to be sure that his code modifications will work for every execution of the program. This is an especially useful property to protect against software cracking, where adversaries aim to widely distribute modified applications.

Parallax implements probabilistically generated function chains by considering each function chain as a series of vectors $v_1, \dots, v_n$, where $v_i \in \{0,1\}^w$ for $1 \le i \le n$. Here, w is the native memory word length in bits (typically 32 or 64). Intuitively, each vector in a function chain corresponds to a gadget address or constant used by the chain (all constants in our function chains are word-sized). Each vector can be generated using a linear combination of vectors from a basis $B = \{b_1, \dots, b_w\}$ which spans the vector space $\{0,1\}^w$.

To support dynamic generation of multiple variants of the same function chain, we define a series of N *index arrays* $A_1, \dots, A_N$, such that each A_i for $1 \le i \le N$ is a two-dimensional array of vector indices. The number of index arrays N can be arbitrarily chosen. If the function chain contains l vectors, then each A_i stores l lists of vector indices. If the l-th vector from the function chain is of gadget type t, then the l-th index list in each A_i contains indices $j_1, \dots, j_k$ which index vectors $b_{j_1}, \dots, b_{j_k}$ from B such that $b_{j_1} \oplus, \dots, \oplus b_{j_k} \in \{g \mid g \text{ implements } t\}$. This means we can form N semantic equivalents for each vector in a function chain by choosing randomly between $A_1, \dots, A_N$ and combining the basis vectors specified there. Figure 4 illustrates this approach to generating function chains. For a function chain of length l, there exist at most N^l variants (assuming that no two A_i store the same index list at any position).

We generate the index arrays by repeatedly compiling the function chain, each time feeding a different gadget mapping to the ROP compiler. By varying the set of gadget addresses used in each mapping, we obtain different compiled variants of the function chain. We then split each vector from every compiled variant into basis vectors, and store the indices of these in the index arrays. Note that the compiled function chains themselves are not stored in the binary, as shown in Figure 4. Instead, we use the index arrays to probabilistically generate a function chain variant at runtime, just before the function chain is called. Since we randomly choose between the index arrays at vector granularity, the possible number of function chain variants generated at runtime is greater than the number of compiled variants.

In Section VII-B, we discuss our performance experiments on dynamic function chain generation. We report results for function chains encrypted with RC4 and xor, and for probabilistically generated function chains.

C. Instruction-Level Verification

In addition to function-level verification, we also experimented with instruction-level verification. Instead of translating a whole function, this approach translates many single instructions into short ROP chains, which we refer to as μ-chains. Figure 3b compares μ-chains to function chains.

We find μ-chains to be suboptimal for several reasons. (1) To minimize control transfer overhead, μ-chains are best implemented inline in the code section, as shown in Figure 3b. This means that, unlike function chains, μ-chains cannot benefit from additional protection by checksumming (due to the attack of Wurster et al. [36]) or self-modification. (2) The inline gadget setup instructions used by μ-chains can be detected through static analysis, and can be exploited by an adversary to pinpoint gadgets used for protection. (3) The overhead of μ-chains exceeds that of function chains by a factor of $2\times$ on average, because each μ-chain contains its own prologue and epilogue. For these reasons, we focus on function-level verification in this paper.

VI. Attack Resistance

This section discusses the resistance of our technique to attacks which attempt to disable, circumvent, or tamper with the verification code. As mentioned, the verification code is a translation to ROP of code from the original program, which is required for the program to correctly execute. The challenge for an adversary is thus to tamper with the protected program in such a way that this is not detected by the verification code, without modifying the verification code functionality. The rest of this section discusses three attack classes.

A. Code Restoration

An adversary may attempt to evade detection by restoring modified code after it has executed. Such code restore attacks are only relevant in dynamic (runtime) tampering. For static attack scenarios, as in software cracking, adversaries cannot use code restore attacks. It is well-recognized in the literature that no self-sufficient tamperproofing algorithm can completely prevent code restore attacks [14]. However, *Parallax* complicates such attacks in several ways. (1) It is critical to use verification functions which are executed repeatedly through the runtime of the protected application. As we show in Section VII-B, *Parallax* achieves this while keeping performance overhead low (up to 4%). (2) By decoupling verification code from protected code, *Parallax* maximizes the difficulty for an adversary to pinpoint which modifications trigger the tamper response.

B. Verification Code Replacement

Additionally, an adversary may tamper with the code locations where verification code is initialized, and attempt to replace it with another ROP chain, or with non-ROP code. Several factors prevent such attacks. (1) The replacement code must be functionally equivalent to the verification code, while not using the same gadgets. The requirement for functional equivalence imposes a first challenge to the adversary, namely the need to reverse engineer the verification code. This is a time-consuming effort, which is complicated by the lack of analysis tools for ROP code [26]. (2) More fundamentally, *Parallax* increases the reverse engineering effort by using dynamically generated and self-modifying ROP code, as proposed in Section V-B. (3) Because the verification code initialization is deterministic, it could be protected using techniques orthogonal to ours, like oblivious hashing.

C. Verification Code Modification

Adversaries may also modify the verification code itself. Here, one of the main strengths of *Parallax* becomes apparent: because the verification code resides in data memory, it can be protected by any traditional checksumming technique. At the same time, there is no risk of the attack of Wurster et al. [36], because that attack relies on the handling of code as data. To prevent persistent tampering with the checksumming code, we propose to use a network of cross-verifying checksums, as explored by Chang et al. for code verification [11]. Such a network can be implemented by embedding the checksumming code inside the verification functions, and letting each verification function checksum itself as well as several others. This way, checksumming can also be embedded in dynamically generated verification code (which itself also complicates tampering). As checksumming is not fundamental to our technique, we leave its implementation to future work. We expect that the performance of checksumming will be similar to that of verification code encryption (evaluated in Section VII-B).

VII. Evaluation

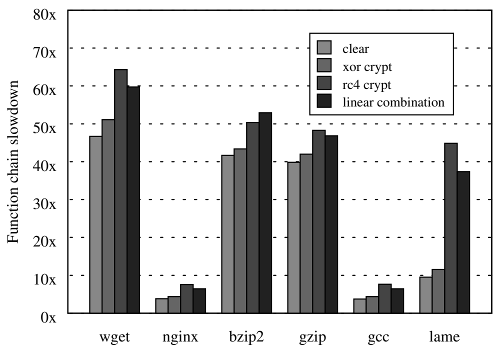
This section evaluates the performance of *Parallax*, our prototype implementation of ROP-based code integrity verification. In Section VII-A, we measure what percentage of code bytes in real-world programs can be protected using overlapping gadgets. Next, Section VII-B evaluates the runtime overhead induced by the verification code using the gadgets.

A. Protectable Code Locations

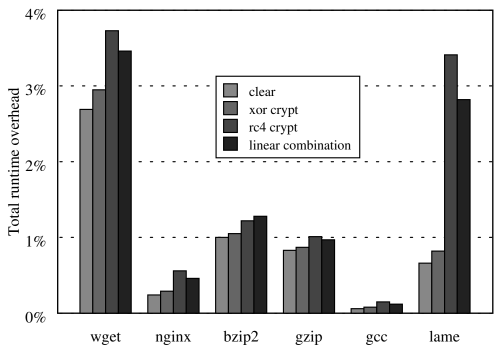
We define a *protectable code byte* as an instruction byte for which we can craft an overlapping gadget using one of the rewriting rules discussed in Section IV-B. We used *Parallax* to measure the percentage of protectable code bytes in a set of real-world programs consisting of wget, nginx, bzip2, gzip, gcc, and lame, compiled for x86 using gcc 4.6.3.

Figure 6 shows the results of our experiment. The figure shows the percentage of protectable code bytes using existing near-return gadgets, far-return gadgets, and gadgets created by modifying immediates and jump offsets. Additionally, the figure shows the percentage of code bytes that can be protected using any of these rules. This percentage can be lower than the sum of the per-rule percentages, since some code bytes can be protected using multiple rules.

In our experiments, modifications to immediates were only applied in `add`, `adc`, `sub`, `sbb`, and `mov` instructions. Examples of how we apply such modifications were discussed



(a) Function chain slowdowns.



(b) Whole-program overhead for function chains.

Fig. 5: Slowdowns and whole-program overheads for function chains.

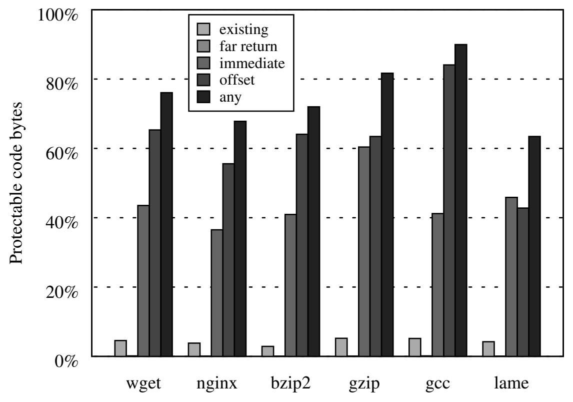


Fig. 6: Code bytes protectable by rules from Section IV-B.

in Section IV-B. Modifications to jump offsets were considered for all variants of the `jmp` and `jcc` instructions, as well as for `call` instructions. No results are shown for the spurious-instructions rule, as this rule can always be applied. Furthermore, we limited the length of the considered gadgets to six instructions, as longer gadgets are difficult to use in practical ROP chains. Note that it is not necessarily possible to protect all potentially protectable code bytes at once, since the required modifications may conflict.

The lowest protectability rate was 63% for lame, and the highest rate was 90% for gcc. Using any of the rewriting rules, an average of 75% of the code bytes is protectable. As can be seen from Figure 6, between 3% and 6% of the code bytes contains an existing overlapping near-return gadget. Additionally, up to 1% of the code bytes in the test programs overlaps with a far-return gadget. The near-return and far-return gadgets add up to protect between 4% and 7% of the code bytes, without requiring any modifications. The protectability rate for the immediate-modification rule ranges from 37%–60%, while ranging from 43%–84% for jump-modification.

B. Runtime Overhead

We also evaluated the performance of verification code. To evaluate the performance, we selected one function from each program and measured the performance before and after translating it to ROP code. We use the following (fully automatable) algorithm to select which function to translate in a given program. (1) We first analyze the call graph of the program to find functions which are called repeatedly from several locations. This ensures that the integrity is verified repeatedly. (2) We then profile the program, and select the functions from the previous step which contribute less than a threshold to the total execution time (2% in our experiments). (3) Finally, we select from this the function containing the most types of operations, ensuring good coverage of all gadgets. We considered both application-specific and library functions for translation to function chains.

For each function, we measured the cleartext slowdown induced purely by the transformation to a function chain. Furthermore, we measured slowdowns for RC4-encrypted and xor-encrypted function chains, as well as function chains generated probabilistically through linear combination (as described in Section V-B). Figures 5a and 5b show the resulting function chain slowdowns and overall runtime impacts for each of these hardening strategies.

The cleartext function chain slowdown ranges from 3.7$\times$ for gcc to 46.7$\times$ for wget. RC4-encrypted function chains have the poorest performance, followed by probabilistically generated and xor-encrypted function chains. The slowdown of RC4-encrypted function chains ranges from 7.6$\times$ for nginx to 64.3$\times$ for wget, but the greatest performance decrease is seen in lame. This is because the function chain for this test case executes in only 4μs, so that the RC4 initialization phase causes a large slowdown.

Despite the significant slowdown induced on each translated function, the whole-program overheads are limited, ranging from 0.1% for gcc to 2.7% for wget using cleartext function chains. When using RC4 encryption, the overhead ranges from 0.2% for gcc to 3.7% for wget. In our experiments, the decryption step (xor, RC4, or linear combination) was performed on each function chain call. Summarizing, the overall runtime overhead of protected binaries is limited, provided that care is taken not to use performance-critical functions as verification code.

VIII. Discussion and Limitations

This section discusses the tradeoffs and limitations of *Parallax*. We also compare these tradeoffs to those of other tamperproofing techniques.

A. Dynamic Circumvention

The goal of our work is to protect code against explicit modifications. Some dynamic analysis primitives, such as software breakpoints and dynamic code patching, are also detected by *Parallax* (see Section IV-A). However, *Parallax* does not explicitly defend against dynamic analysis. Specifically, some dynamic analysis tools, such as Pin [1] and DynamoRIO [8], instrument binaries without altering their runtime code section, and are thus not detected by *Parallax*. However, *Parallax* can protect specialized detection code for these tools, which was developed in related work [16].

B. Control Flow Integrity

Prior work has explored the detection of ROP-based exploit code at runtime, using heuristic-based system-level monitoring tools like kBouncer and ROPGuard [27, 28]. These tools may conflict with our tamperproofing algorithm, detecting its use of ROP code as if it were malicious. However, recent work has shown that heuristic-based monitoring approaches can be fundamentally circumvented by simple modifications to ROP chains [15, 19, 31]. *Parallax* can employ these same modifications to avoid conflicts. For instance, using a small number of long gadgets or NOP-gadgets is sufficient to allow *Parallax* to operate in unison with heuristic system-level ROP-monitoring tools [19]. Since such gadgets are present by nature in nearly all applications, *Parallax* can use them without opening the application up to ROP attacks any more than it already was.

Stronger Control Flow Integrity (CFI) approaches like CCFIR [38] and the original work by Abadi et al. [4] are applied at the application level rather than the system level. These approaches incur higher overhead than system-level approaches, and are difficult to apply to legacy binaries, which has thus far prevented their widespread deployment. Full CFI, as proposed by Abadi et al., is difficult to combine with *Parallax*, due to its need to record (and thus reveal to an adversary) all legal control transfers. However, as these are application-level approaches, there is a large amount of leeway for balancing the level of CFI enforcement against the desired level of tamperproofing per binary.

C. Protection Coverage

Our technique provides vastly different protection tradeoffs than oblivious hashing. (1) As mentioned, oblivious hashing can only protect code with deterministic execution state. Unprotectable code includes code which depends on system calls like those in the ptrace detector from Section IV-A [13]. Arguably, such non-deterministic code is more likely to be targeted by adversaries than deterministic code. For instance, adversaries commonly modify control flow instructions which depend on external inputs like license keys. Thus, a significant advantage of our technique is that it can protect both deterministic and non-deterministic code. (2) Oblivious hashing covers only code paths of which the state was recorded during testing. In contrast, our technique is completely static, and can be applied even to untested code.

Indefinite attack resistance is impossible to implement in a self-sufficient tamperproofing system [5]. Rather, *Parallax* is designed to raise the bar for attackers, and increase the effort required to tamper with protected code. A determined adversary may eventually succeed in tampering with code by ensuring one or more of the following conditions. (1) The modifications reside entirely in instructions without overlapping gadgets. As discussed in Section VII, *Parallax* attempts to minimize such instructions. (2) Protected code is modified such that the resulting gadgets do not affect the outcome of the verification code. (3) Protected code is altered such that the resulting gadgets are semantically equivalent to the originals (including memory/register allocation). These conditions significantly restrict the modifications that can be safely made, making it much harder for an attacker to implement arbitrary modifications.

IX. Related Work

To the best of our knowledge, no prior work exists on using Return-Oriented Programming techniques for tamperproofing. Additionally, in contrast to our work, previous work on tamperproofing does not discuss how the proposed techniques can be applied at the binary level. We therefore believe that our work is the first to discuss code protection which can be implemented completely at the binary level, and can thus be used to tamperproof legacy binaries.

Traditional anti-tampering algorithms make use of code introspection, typically in the form of checksumming [14]. A highly resilient example of such an algorithm was proposed by Chang et al. [11], who use a network of cross-verifying code regions based on checksumming. Unfortunately, Wurster et al. have shown all such algorithms to be inherently vulnerable to an attack which exploits the distinct handling of code and data in modern processors [36]. The attack completely defeats all introspection-based algorithms by allowing an attacker to freely modify code in the processor's instruction cache, while leaving the data cache untouched. Later work has explored methods to re-enable code self-checksumming by implementing checks to detect the attack of Wurster et al [18]. Unfortunately, these checks require W$\oplus$X protection to be disabled, making the checksummed binary vulnerable to traditional code injection.

The foremost among the few algorithms designed to defeat this attack is oblivious hashing [13, 20]. OH verifies code integrity by checking that hashes of the execution state correspond to known correct values. In principle, it provides strong protection which is difficult to circumvent. However, the execution state is required to be deterministic, preventing OH from protecting code with non-deterministic inputs, like environment parameters or system call return values. The main benefit of our approach compared to OH is that it can protect code regions which OH cannot.

Previous work has proposed overlapping non-gadget instructions for tamperproofing [20, 24]. Instruction-level overlapping is only applicable to architectures with variable-length byte-aligned instructions [20]. In contrast, our ROP-based approach does not have this restriction [10, 12]. Furthermore, overlapping non-gadget instructions requires the insertion of additional jumps and partial instructions in the protected code, which leads to whole-program slowdowns of up to 3$\times$ [20]. Our approach provides better overall performance, and can keep performance overhead isolated from the protected code itself. Another approach to overlapping is to share common code blocks between functions. The usefulness of this approach is limited, as most common code blocks found in real-world binaries are non-sensitive instruction sequences like function prologues. It is typically not possible to protect non-trivial code blocks longer than one instruction using this approach [20].

Concurrently with our work, Lu et al. have explored the use of ROP for code obfuscation [25]. However, they do not consider tamperproofing, and thus do not explore how to maximize the coverage of protective gadgets, or how to craft gadgets which overlap with sensitive instructions. Instead, their work focuses on the use of existing (partial) gadgets to create ROP chains which are embedded with the intent of hiding functionality. Furthermore, Lu et al. do not attempt to prevent adversaries from tampering with their ROP chains once these are discovered. Similarly, prior work has proposed code hiding techniques based on function reuse, but this work has not focused on extending this to tamperproofing [23].

X. Conclusion

We introduced a novel code self-verification technique based on overlapping ROP gadgets with selected code. Several rewriting rules can be used to increase the coverage of protective gadgets, such that up to 90% of all code bytes are protectable. This coverage exceeds that of oblivious hashing, and our technique provides better protection for commonly attacked non-deterministic control flow instructions. Unlike code introspection-based verification algorithms, our approach is not vulnerable to direct instruction cache modification attacks. Furthermore, in contrast to oblivious hashing algorithms, our approach can protect non-deterministic code. The performance overhead of our approach can be confined to verification code which is separate from the protected code. Thus, performance-sensitive code is protectable without any slowdown, confining the performance penalty to other code. The performance overhead for programs protected using our technique is less than 4%.

Acknowledgements

This work was supported by the European Research Council Starting Grant "Rosetta", the European Commission EU FP7-ICT-257007 SysSec project, and the Netherlands Organisation for Scientific Research (NWO).

References

[1] Intel Pin. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[2] ROPC compiler. https://github.com/pakt/ropc.

[3] A Framework for Aviation Cybersecurity, 2013. Technical report, AIAA.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of Conference on Computer and Communications Security*, CCS'05, 2005.

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Crypto'01*, 2001.

[6] P. Biondi and F. Desclaux. Silver Needle in the Skype. In *Black Hat Europe*, 2006.

[7] J. Borello and L. Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal of Computer Virology and Hacking Techniques*, 2008.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of Symposium on Code Generation and Optimization*, CGO'03, 2003.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of Conference on Computer-Aided Verification*, CAV'11, 2011.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of Conference on Computer and Communications Security*, CCS'08, 2008.

[11] H. Chang and M. J. Atallah. Protecting Software Code by Guards. In *Proceedings of Digital Rights Management Symposium*, DRM'01, 2001.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In *Proceedings of Conference on Computer and Communications Security*, CCS'10, 2010.

[13] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In *Proceedings of ACM Workshop on Information Hiding and Multimedia Security*, IH'05, 2003.

[14] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[15] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of USENIX Security Symposium*, USENIX Sec'14, 2014.

[16] F. Falcon and N. Riva. Dynamic Binary Instrumentation Frameworks: I Know You're There Spying On Me. In *RECON'12*, 2012.

[17] N. Falliere, L. O Murchu, and E. Chien. W32.Stuxnet Dossier, 2011. Technical report, Symantec.

[18] J. Giffin, M. Christodorescu, and L. Kruger. Strengthening Software Self-Checksumming via Self-Modifying Code. In *Proceedings of Annual Computer Security Applications Conference*, ACSAC'05, 2005.

[19] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of IEEE Symposium on Security and Privacy*, S&P'14, 2014.

[20] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards Integral Binary Execution: Implementing Oblivious Hashing Using Overlapped Instruction Encodings. In *Proceedings of Conference on Multimedia and Security*, MM&Sec'07, 2007.

[21] Kaspersky Lab Global Research and Analysis Team. Gauss: Abnormal Distribution, 2012. Technical report, Kaspersky.

[22] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of Symposium on Performance Analysis of Systems and Software*, ISPASS'10, 2010.

[23] Z. Lin, X. Zhang, and D. Xu. Reuse-Oriented Camou-

flaging Trojan: Vulnerability Detection and Attack Construction. In *Proceedings of Conference on Dependable Systems and Networks*, DSN'10, 2010.

[24] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of Conference on Computer and Communications Security*, CCS'03, 2003.

[25] K. Lu, S. Xiong, and D. Gao. RopSteg: Program Steganography with Return Oriented Programming. In *Proceedings of ACM Conference on Data and Application Security and Privacy*, CODASPY'14, 2014.

[26] K. Lu, D. Zou, W. Wen, and D. Gao. deRop: Removing Return-Oriented Programming from Malware. In *ACSAC'11*, 2011.

[27] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of USENIX Security Symposium*, USENIX Sec'13, 2013.

[28] M. Polychronakis and A. D. Keromytis. ROP Payload Detection Using Speculative Code Execution. In *MALWARE'11*, 2011.

[29] K. A. Roundy and B. P. Miller. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 2012.

[30] H. Saïdi, V. Yegneswaran, and P. Porras. Experiences in Malware Binary Deobfuscation. *Virus Bulletin*, 2010.

[31] F. Schuster, T. Tendyck, J. Pewny, A. Maass, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of Symposium on Research in Attacks, Intrusions and Defenses*, RAID'14, 2014.

[32] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of USENIX Security Symposium*, USENIX Sec'11, 2011.

[33] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (on the x86). In *Proceedings of Conference on Computer and Communications Security*, CCS'07, 2007.

[34] A. Srivastava and J. Giffin. Automatic Discovery of Parasitic Malware. In *Proceedings of Symposium on Research in Attacks, Intrusions and Defenses*, RAID'10, 2010.

[35] United States Department of Defense. DoD Software Protection Initiative. In *SSTC'06*, 2006. Description at http://sstc-online.org/2006/index.cfm?fs=exhlist&Letter=D.

[36] G. Wurster, P. van Oorschot, and A. Somayaji. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *Proceedings of IEEE Symposium on Security and Privacy*, S&P'05, 2005.

[37] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of Network and Distributed System Security Symposium*, NDSS'08, 2008.

[38] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, S&P'13, 2013.