Defending against Return-Oriented Programming

Vasileios Pappas

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Defending against Return-Oriented Programming

Vasileios Pappas

Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections. ROP exploits are facilitated mainly by the lack of complete address space randomization coverage or the presence of memory disclosure vulnerabilities, necessitating additional ROP-specific mitigations. Existing defenses against ROP exploits either require source code or symbolic debugging information, or impose a significant runtime overhead, which limits their applicability for the protection of third-party applications.

We propose two novel techniques to prevent ROP exploits on third-party applications without requiring their source code or debug symbols, while at the same time incurring a minimal performance overhead. Their effectiveness is based on breaking an invariant of ROP attacks: knowledge of the code layout, and a common characteristic: unrestricted use of indirect branches. When combined, they still retain their applicability and efficiency, while maximizing the protection coverage against ROP.

The first technique, *in-place code randomization*, uses narrow-scope code transformations that can be applied statically, without changing the location of basic blocks, allowing the safe randomization of stripped binaries even with partial disassembly coverage. These transformations effectively eliminate 10%, and probabilistically break 80% of the useful instruction sequences found in a large set of PE files. Since no additional code is inserted, in-place code randomization does not incur any measurable runtime overhead, enabling it to be easily used in tandem with existing exploit mitigations such as address space layout randomization. Our evaluation using publicly available ROP exploits and two ROP code generation toolkits demonstrates that our technique prevents the exploitation of the tested vulnerable Windows 7 applications, including Adobe Reader, as well as the automated construction of alternative ROP payloads that aim to circumvent in-place code randomization using solely any remaining unaffected instruction sequences.

The second technique is based on the detection of abnormal control transfers that take place during ROP code execution. This is achieved using hardware features of commodity processors, which incur negligible runtime overhead and allow for completely transparent operation without requiring any modifications to the protected applications. Our implementation for Windows 7, named *kBouncer*, can be selectively enabled for installed programs in the same fashion as user-friendly mitigation toolkits like Microsoft's EMET. The results of our evaluation demonstrate that kBouncer has low runtime overhead of up to 4%, when stressed with specially crafted workloads that continuously trigger its core detection component, while it has negligible overhead for actual user applications. In our experiments with in-the-wild ROP exploits, kBouncer successfully protected all tested applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader.

In addition, we introduce a technique that enables ASLR for executables with stripped relocation information by incrementally adjusting stale absolute addresses at runtime. The technique relies on runtime monitoring of memory accesses and control flow transfers to the original location of a module using page table manipulation. We have implemented a prototype of the proposed technique for Windows XP, which is transparently applicable to third-party stripped binaries. Our results demonstrate that incremental runtime relocation patching is practical, incurs a runtime overhead of up to 83% in most of the cases for initial runs of protected programs, and has a low runtime overhead of 5% on subsequent runs.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor Angelos D. Keromytis for giving me the opportunity and supporting me throughout my graduate studies. I honestly could not hope for a better advisor-student relationship.

My friend and close collaborator Michalis Polychronakis deserves a big thank you. His guidance was an essential ingredient for the success of this work.

I would like to thank all the members of the NSL group, and especially Vasileios P. Kemerlis, for making our lab a fun place to work. I would like to thank everyone in the CS department that helped along the way.

Finally, nothing would be possible without the support, dedication and love of Eleni and my family.

Chapter 1

Introduction

1.1 Motivation

Attack prevention technologies based on the No eXecute (NX) memory page protection bit, which prevent the execution of malicious code that has been injected into a process, are now supported by most recent CPUs and operating systems [Miller *et al.*, 2011]. The wide adoption of these protection mechanisms has given rise to a new exploitation technique, widely known as *return-oriented programming* (ROP) [Shacham, 2007], which allows an attacker to circumvent non-executable page protections without injecting any code. Using return-oriented programming, the attacker can link together small fragments of code, known as *gadgets*, that already exist in the process image of the vulnerable application. Each gadget ends with an indirect control transfer instruction, which transfers control to the next gadget according to a sequence of gadget addresses injected on the stack or some other memory area. In essence, instead of injecting binary code, the attacker injects just data, which include the addresses of the gadgets to be executed, along with any required data arguments.

Several research works have demonstrated the great potential of this technique for bypassing defenses such as read-only memory [Checkoway *et al.*, 2009], kernel code integrity protections [Hund *et al.*, 2009], and non-executable memory implementations in mobile devices [Dullien *et al.*, 2010] and operating systems [Zovi, 2010b; Solé, 2010; Zovi, 2010a; Vreugdenhil, 2010]. Consequently, it was only a matter of time for ROP to be employed in real-world attacks. Recent exploits against popular applications use ROP code to bypass exploit mitigations even in the latest OS versions, including Windows 7 SP1. ROP exploits are included in the most common exploit packs [Baumgartner, 2010; Parkour, 2011], and are actively used in the wild for mounting drive-by download attacks.

Attackers are able to a priori pick the right code pieces because parts of the code image of the vulnerable application remain static across different installations. Address space layout randomization (ASLR) [Miller *et al.*, 2011] is meant to prevent this kind of code reuse by randomizing the locations of the executable segments of a running process. However, in both Linux and Windows, parts of the address space do not change due to executables with fixed load addresses [Fresi Roglia *et al.*, 2009], or shared libraries incompatible with ASLR [Zovi, 2010b]. Furthermore, in some exploits, the base address of a DLL can be either calculated dynamically through a leaked pointer [Li, 2011; Vreugdenhil, 2010; Serna, 2012], or brute-forced [Shacham *et al.*, 2004].

Other defenses against code-reuse attacks complementary to ASLR include compiler extensions [Li *et al.*, 2010; Onarlioglu *et al.*, 2010], code randomization [Forrest *et al.*, 1997; Bhatkar *et al.*, 2005; Kil *et al.*, 2006], control-flow integrity [Abadi *et al.*, 2005], and runtime solutions [Davi *et al.*, 2011; Chen *et al.*, 2009; Davi *et al.*, 2009]. In practice, though, most of these approaches are almost never applied for the protection of the COTS software currently targeted by ROP attacks, either due to the lack of source code or debugging information, or due to their increased overhead. In particular, from the above techniques, those that operate directly on compiled binaries, e.g., by permuting the order of functions [Bhatkar *et al.*, 2005; Kil *et al.*, 2006] or through binary instrumentation [Abadi *et al.*, 2005], require precise and complete extraction of all code and data in the executable sections of the binary. This is possible only if the corresponding symbolic debugging information is available, which however is typically stripped from production binaries. On the other hand, techniques that do work on stripped binary executables using dynamic binary instrumentation [Davi *et al.*, 2011; Chen *et al.*, 2009; Davi *et al.*, 2009], incur a significant runtime overhead that limits their adoption.

1.2 Requirements and Goals

From the discussion above it is clear that there is a need for more practical ROP defenses. The three main requirements we consider essential for a practical ROP defense are:

- **Low performance overhead.** Defense solutions that impose a significant slowdown rarely see any wide adoption. Of course, such solutions might be valuable in high-security environments with ample resources. However, for our work, we consider performance overhead a critical factor.
- **Applicability to third-party applications.** Requiring access to the source code of legacy applications in order to retrofit a defense solution might be a difficult task, if not impossible in some cases. Even worse, redistributing recompiled versions of the protected applications poses its own challenges. Thus, protecting third-party applications in the absence of source code or debug symbols becomes very important.
- **Effectiveness.** At a minimum, a ROP defense should be able to protect against current real-world attacks. In addition, a technique that does not provide complete protection should significantly raise the bar for attackers. For example, a technique that prevents current attacks, but, can be circumvented by trivially altering the attack payload, is of lesser value.

Aiming to fulfil the requirements above, in this work we propose two ROP defense techniques that use different approaches to achieve the same goals.

We first focused on improving the current state-of-the-art in techniques that can be statically applied. Towards that end, we designed a fine-grained randomization scheme, named in-place code randomization [Pappas *et al.*, 2012]. Knowledge of the code layout (i.e., locations of instructions) and environment (i.e., OS APIs, system call table, etc.) is an invariant for ROP attacks, and this is exactly what in-place code randomization attacks. Compared to previous approaches, the key difference of our technique is the ability to randomize stripped binaries. This is achieved by modifying only the code that can be safely extracted from compiled binaries, while preserving the length of the randomized instructions and basic blocks — that way, the semantics of undiscovered code parts are preserved. In addition, since code length remains the same during the randomization (i.e., no new instructions are inserted), there is practically no runtime performance overhead.

As stated before, current defenses that offer dynamic, runtime protection are based on dynamic binary instrumentation, which imposes a prohibitively high performance overhead. The goal we set for our second technique was to improve the protection offered by dynamic, runtime approaches, while at the same time, lowering the performance overhead to a minimal level. In addition, we introduced a forth requirement:

- **Transparency.** Protecting applications without changing a single bit of them not only eliminates the need for any code structure assumptions, but also enables compatibility with other protections, like code-signing, etc.

Our second technique, indirect branch tracing [Pappas *et al.*, 2013], detects the execution of ROP code at runtime by identifying abnormal control flow transfers. Our technique prohibits the arbitrary use of indirect branches, which is a characteristic of ROP attacks — this is how smaller fragments of code are linked together. To satisfy the transparency requirement and keep performance overhead to a minimal level, it uses hardware features found on commodity processors to monitor indirect branches. Indirect branch tracing is the first technique to satisfy all four requirements.

Our proposed techniques target different ROP invariants and characteristics (knowledge of code layout vs unrestricted indirect branches). Also, they are applied in different ways (statically vs dynamically at runtime). This means that these two techniques are both compatible and, moreover, complemenentary to each other.

In addition to the two defense techniques above, we also propose a technique to dynamically reconstruct relocation information for stripped binaries [Pappas *et al.*, 2014]. The availability of relocation information for ROP protection is very important for two reasons: not only does it enable ASLR, which would otherwise be impossible, but it also improves disassembly accuracy, which in turns improves the coverage of in-place code randomization, or other similar schemes.

In summary, the hypothesis of this thesis is the following:

Using in-place code randomization and indirect branch tracing to break the code layout knowledge invariant and unrestricted indirect branching characteristic of ROP, provides a practical solution in terms of efficiency, deployability and effectiveness.

1.3 Contributions

- We introduced in-place code randomization, a novel and practical approach for hardening third-party software against ROP attacks. We describe in detail various narrow-scope code transformations that do not change the semantics of existing code, and which can be safely applied on compiled binaries without symbolic debugging information.
  - We implemented in-place code randomization for x86 PE executables, and experimentally verified the safety of the applied code transformations and their practically zero performance overhead with extensive runtime code coverage tests using third-party executables.
  - We provide a detailed analysis of how in-place code randomization affects available gadgets using a large set of 5,235 PE files. On average, the applied transformations effectively eliminate about 10%, and probabilistically break about 80% of the gadgets in the tested files.
  - We evaluate in-place code randomization using publicly available ROP exploits and generic ROP payloads, as well as two ROP payload construction toolkits. In all cases, the randomized versions of the executables break the malicious ROP code, and prevent the automated construction of alternative payloads using the remaining unaffected gadgets.
- We developed a practical and transparent ROP exploit mitigation technique based on runtime tracing of indirect branch instructions using the LBR feature of recent CPUs.
  - We have implemented the branch tracing approach as a self-contained toolkit for Windows 7 and describe in detail its design and implementation.

  - We provide a quantitative analysis of the robustness of the proposed ROP code execution prevention technique against potential evasion attempts.
  - We have experimentally evaluated the performance and effectiveness of branch tracing for ROP prevention, and demonstrate that it can prevent in-the-wild exploits against popular applications with negligible runtime overhead.

- We devised a technique for dynamically reconstructing missing relocation information from stripped binaries. Our technique can be used to enable forced ASLR or or resolve base address conflicts for third-party non-relocatable binaries.

  - We have implemented the proposed approach as a self-contained software hardening tool for Windows applications, and describe in detail its design and implementation.
  - We have experimentally evaluated the performance and correctness of our approach using standard benchmarks and popular applications, and demonstrate its effectiveness.

1.4 Dissertation Roadmap

Chapter 2 covers some background information about ROP and reviews related work. Chapter 3 and 4 describe the design, implementation and experimental evaluation of in-place code randomization and indirect branch tracing, respectively—at the end of Chapter 4 we also explore the benefits of combining the two techniques. Dynamic relocation reconstruction is described in Chapter 5. Finally, Chapter 6 presents the conclusions draw from this work, along with some future work directions.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Code-reuse Attacks Evolution

The introduction of non-executable memory page protections in popular OSes, even for CPUs that do not support the No eXecute (NX) bit, led to the development of the return-to-libc exploitation technique [Designer, 1997]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping to the beginning of a library function such as `system()`, the attacker can for example spawn a shell without the need to inject any code.

Frequently though, especially for remote exploitation, calling a single function is not enough. In these cases, multiple return-to-libc calls can be "chained" together by ensuring that before returning from one function to the next one, the stack pointer has been correctly adjusted to the beginning of the prepared stack frame for the next call. For instance, for a function with two arguments, this can be achieved by first returning to a short instruction sequence such as `pop reg; pop reg; ret;`found anywhere within the executable part of the process image [Newsham, 2000; Nergal, 2001]. The `pop` instructions adjust the stack pointer beyond the arguments of the previously executed function (one `pop` for each argument), and then `ret` transfers control to the next chained function. This approach, however, is not applicable in cases where the function arguments need to be passed through registers. Sebastian Krahmer introduced the “borrowed code chunks” technique [Krahmer, 2005], In that case, a few short instruction sequences ending with a `ret` instruction can be chained directly to set the proper registers with the desired arguments, before calling the library function.

In the above code-reuse techniques, the executed code consists of one or a few short instruction sequences followed by a large block of code belonging to a library function. Hovav Shacham demonstrated that using only a carefully selected set of short instruction sequences ending with a `ret` instruction, known as *gadgets*, it is possible to achieve arbitrary computation, obviating the need for calling library functions [Shacham, 2007]. This powerful technique, dubbed *return-oriented programming*, in essence gives the attacker the same level of flexibility offered by arbitrary code injection without injecting any code at all—the injected payload comprises just a sequence of gadget addresses intermixed with any necessary data arguments.

In a typical ROP exploit, the attacker needs to control both the program counter and the stack pointer: the former for executing the first gadget, and the latter for allowing its `ret` instruction to transfer control to subsequent gadgets. Depending on the vulnerability, if the ROP payload is injected in a memory area other than the stack, e.g., the heap, then the stack pointer must first be adjusted to the beginning of the payload through a stack pivot [Erlingsson, 2007; Zovi, 2010b]. In a follow up work [Checkoway *et al.*, 2010], Checkoway et al. demonstrated that the gadgets used in a ROP exploit need not necessarily end with a `ret` instruction, but with any other indirect control transfer instruction. This also allows the use of any general purpose register in place of the stack pointer as an “index” register for controlling the execution of the gadgets, bypassing any protections based on stack integrity.

The ROP code used in recent exploits against Windows applications is mostly based on gadgets ending with `ret` instructions, which conveniently manipulate both the program counter and the stack pointer, although a couple of gadgets ending with `call` or `jmp` are also used for calling library functions. In all publicly available Windows exploits so far, attackers do not have to rely on a fully ROP-based implementation for the whole malicious

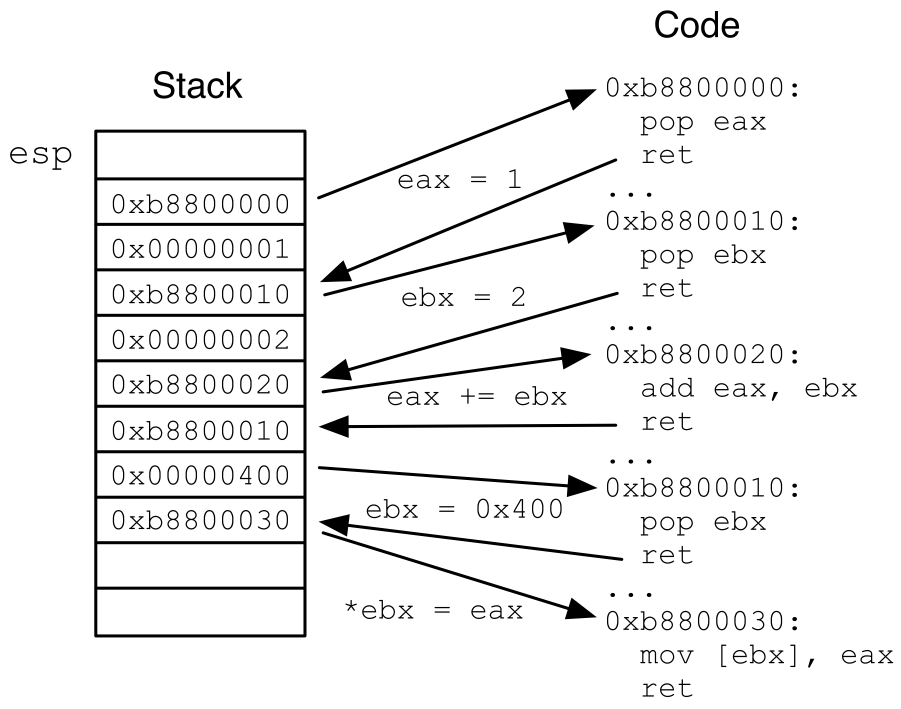


Figure 2.1: Example a simple ROP program that adds two numbers and stores the result in a given memory address. (Note that in order to keep diagram simpler, the gadget at `0xbb800010` appears two times in the code column.)

code that needs to be executed, after triggering a memory corruption vulnerability. Instead, ROP code is used only as a first stage for bypassing DEP [Miller *et al.*, 2011]. Typically, once control flow has been hijacked, the ROP code allocates a memory area with write and execute permissions by calling a library function like `VirtualAlloc`, copies into it some plain shellcode included in the attack vector, and finally jumps to the copied shellcode which now has execute permission [Erlingsson, 2007].

2.1.2 ROP Example

To better demonstrate how return-oriented programming works in practice, we provide of a simple program that performs a basic operation: adding two input numbers and storing the result in a given memory location. Figure 2.1 depicts a graphical representation of our example for x86. The runtime stack of the vulnerable application is on the left side and its contents are controllable by the attacker. On the right side, we see small fragments of code (or, *gadgets*) that belong to the executable segment of the same vulnerable application. It is important to note that these instruction sequences could either be part of the application's intended executable code (i.e., generated by the compiler), or, correspond to unintended overlapping instructions as a results of the variable instruction length of x86. An invariant that holds for all ROP attacks is that the attacker has to have knowledge of the code layout. The arrows between the stack and the gadgets illustrate how control flows between each gadget, based on the contents of the stack. Also, the results of executing each gadget appear between the control flow arrows.

Assuming that the attacker has filled out the stack with the values shown in Figure 2.1 and that a return instruction is executed, control flow is going to be transferred to address `0xb8800000`, which is the beginning of the first gadget. The `pop eax` instruction loads the next value on the top of the stack in the `eax` register. The `ret` which follows effectively links this gadget with the second one, which is located at address `0xb8800020`, by transferring control to the address found on the top of the stack. The use of indirect branches to link gadgets together is one of the main characteristics of ROP. Similarly, the second gadget loads another value from the stack to the `ebx` register. Again, its `ret` instruction links it with the next gadget by using the address on the current top of the stack. At this point, `eax` and `ebx` contain the values 1 and 2, respectively. The third gadget, found at address `0xb8800020`, adds the contents of these two registers. Then, the second gadget is used again to load yet another value from the stack in `ebx`. The content of `ebx` is used as the destination address by the last gadget to store the result of the addition.

What is important to note from this example is the fact that, although no additional code was inserted, the attacker was able to perform a very specific computation. This forms the basis of return-oriented programming and, as stated earlier, it has been shown that, given a certain set of gadgets an attacker has the power to perform arbitrary code execution [Shacham, 2007]. As we see from the example above, the success of a ROP attack depends both on the knowledge of the code layout and the ability to use indirect branches to link gadgets.

Table 2.1 summarizes the invariants and characteristics of ROP attacks. Code-reuse

Table 2.1: Invariants and characteristics of ROP attacks.

Name	Description
Knowledge of code layout	Synthesizing a ROP payload requires access to the code
Unrestricted indirect branching	Arbitrary code fragments are linked through indirect branches

attacks in general depend on having access to the code of the vulnerable application, so the parts to be reused, depending on the attack's goal, can be picked. We identify knowledge of the code layout as an invariant for the success of ROP attacks. After the right code fragments (gadgets) are picked, they have to be linked together to perform the required computation through indirect branches. The availability of unrestricted indirect branches for that reason is a common characteristic of ROP attacks. Coming back to the example above, we see that the attacker needed to have knowledge about the code layout to pick these five gadgets. In addition, there should be no restrictions on the targets of the return instructions, so the gadgets can be linked together.

2.2 Related Work

Almost a decade after the introduction of the return-to-libc technique [Designer, 1997], the wide adoption of non-executable memory page protections in popular OSes sparked a new interest in more advanced forms of code-reuse attacks. The introduction of return-oriented programming [Shacham, 2007] and its advancements [Buchanan *et al.*, 2008; Checkoway *et al.*, 2009; Hund *et al.*, 2009; Checkoway *et al.*, 2010; Dullien *et al.*, 2010; Bletsch *et al.*, 2011b; Schwartz *et al.*, 2011; Solé, 2008; Zovi, 2010b; Zovi, 2010a] led to its adoption in real-world attacks [Baumgartner, 2010; Parkour, 2011]. ROP exploits are facilitated by the lack of complete address space layout randomization in both Linux [Fresi Roglia *et al.*, 2009], and Windows [Zovi, 2010b], which otherwise would prevent or at least hinder [Shacham *et al.*, 2004] these attacks.

In the following subsections we review the related work in the area of ROP defenses. We have divided them in two categories: (i) defenses that break the knowledge of code layout invariant, (ii) defenses that restrict the use of indirect branches. The following subsection

Table 2.2: ROP defenses comparison.

Defense	Efficient	Stripped Binaries	Incomplete Disas.	ROP w/o Returns	Transparent	Breaks Layout	Restricts Branches
DROP		✓	✓		✓		✓
DynIMA		✓	✓		✓		✓
ROPdefender		✓	✓		✓		✓
Return-less	✓						✓
G-Free	✓			✓			✓
CFL	✓			✓			✓
CFR	✓			✓			✓
ILR	✓	✓		✓		✓	
Binary Stirring	✓	✓		✓		✓	
XIFER	✓	✓		✓		✓	
CCFIR	✓	✓		✓			✓
CFI-COTS	✓	✓		✓			✓
ROPGuard	✓			✓	✓		✓
ROPecker				✓	✓		✓
ROP Sandy	✓	✓	✓		✓		✓
Branch Reg.	✓	✓		✓	✓		✓
CFImon		✓		✓	✓		✓
In-place Rand.	✓	✓	✓	✓		✓	
Ind. Branch Tr.	✓	✓	✓	✓	✓		✓

describes related work that falls into the first category. The next two subsections review defenses that fall under the second category—subdivided based on whether the checks are added statically, or dynamically at runtime. Table 2.2 provides a comparison of this work to these proposals based on whether they: incur performance overhead which is lower than 5%[1] (Efficiency), are applicable on third-party applications (Stripped Binaries), do not require complete extraction of the control flow graph (Incomplete Disas.), protect against ROP that uses gadgets ending in indirect jump/call (ROP w/o Returns), are transparent to the protected applications (Transparent), break the knowledge of code layout ROP invariant (Breaks Layout), attack the unrestricted use of indirect branches ROP characteristic (Restricts Branches).

[1] This is a reasonable requirement for practical purposes, as it was also used in Microsoft's BlueHat Prize Contest [Microsoft, 2012].

2.2.1 Address Space Randomization and Code Diversification

As code-reuse attacks require precise knowledge of the structure and location of the code to be reused, diversifying the execution environment or even the program code itself is a core concept in preventing code-reuse exploits [Cohen, 1993; Forrest *et al.*, 1997]. Address space layout randomization [PaX Team, 2001; Miller *et al.*, 2011] is probably one of the most widely deployed countermeasures against code-reuse attacks. However, its effectiveness is hindered by code segments left in static locations [Fresi Roglia *et al.*, 2009; Zovi, 2010b; Johnson, 2011], while, depending on the randomization entropy, it might be possible to circumvent it using brute-force guessing [Shacham *et al.*, 2004]. Even if all the code segments of a process are fully randomized, vulnerabilities that allow the leakage of memory contents can enable the calculation of the base address of a DLL at runtime [Bennett *et al.*, 2013; Serna, 2012; Li, 2011; Vreugdenhil, 2010; Hund *et al.*, 2013; Snow *et al.*, 2013].

Intra-DLL randomization at the function [Bhatkar *et al.*, 2003; Bhatkar *et al.*, 2005; Kil *et al.*, 2006; Microsoft, b], basic block [Google, 2011; Microsoft, c], or instruction level [Hiser *et al.*, 2012; Wartell *et al.*, 2012; Davi *et al.*, 2013] can provide protection for executables that do not support ASLR, or against de-randomization attacks through memory leaks. The practical deployment of these techniques for the protection of third-party applications depends on the availability of source code [Bhatkar *et al.*, 2003; Bhatkar *et al.*, 2005; Kil *et al.*, 2006; Microsoft, b], debug symbols [Google, 2011; Microsoft, c; Davi *et al.*, 2013], or the accuracy of disassembly and control flow graph extraction [Hiser *et al.*, 2012; Wartell *et al.*, 2012].

At the same time, instruction set randomization (ISR) [Kc *et al.*, 2003; Barrantes *et al.*, 2003] cannot completely prevent code-reuse attacks. Although gadgets that contain overlapping instructions cannot be used under ISR, an attacker can still use all the gadgets that are part of legitimate executable code (i.e., code that is intentionally generated by a compiler). Also, current implementations of ISR rely on heavyweight runtime instrumentation or code emulation frameworks.

2.2.2 Control Flow Integrity and Indirect Branch Protection

The execution of ROP code disrupts the normal call path of typical programs, resulting to an unanticipated flow of control. Control flow integrity [Abadi *et al.*, 2005] can confine program execution within the bounds of a precomputed profile of allowed control flow paths, and thus can prevent most of the irregular control flow transfers that connect the gadgets of a ROP exploit. Depending on program complexity, however, deriving an accurate view of the control flow graph is often challenging. Alternative approaches against return-oriented programming enforce a more relaxed policy for the integrity of indirect control transfers [Onarlioglu *et al.*, 2010; Li *et al.*, 2010; Bletsch *et al.*, 2011a; Pewny and Holz, 2013]. Using code transformations, these techniques eliminate the occurrence of unintended indirect branch instructions in the generated code, and safeguard all legitimate indirect branches using cookies or additional levels of indirection.

The main factor that limits the practical applicability of the above techniques is that they require the recompilation of the target application, which is usually not possible for the popular proprietary applications that are commonly targeted by ROP exploits. To overcome this limitation, more recent proposals introduced ways to enforce control flow integrity on binary programs, without requiring source code or debug symbols [Zhang and Sekar, 2013; Zhang *et al.*, 2013]. As constructing an accurate control flow graph from a striped binary program is an intractable problem, these solutions incorporated looser rules when checking for indirect call or return targets, etc. Although such schemes clearly raise the bar for successful ROP exploitation, it has been shown that, at least in some cases, an attacker can construct a useful ROP payload using allowable indirect control transfers to chain gadgets together [Göktas *et al.*, 2014a].

2.2.3 Runtime Execution Monitoring

Many defenses against return-oriented programming are based on monitoring program execution at the instruction level. A widely used mechanism for this purpose is dynamic binary instrumentation (DBI), using frameworks such as Pin [Luk *et al.*, 2005]. DROP [Chen *et al.*, 2009] and DynIMA [Davi *et al.*, 2009] follow this approach to monitor the frequency of `ret` instructions, and raise an alert in case irregularly many of them are observed within a small window of executed instructions. ROPdefender [Davi *et al.*, 2011] also uses DBI to keep a shadow stack that is updated by instrumenting `call` and `ret` instructions. A disruption of the expected `call`-`ret` pairs due to ROP code is detected by comparing the shadow stack with the system's stack on every function exit. A limitation of the above techniques is that they cannot prevent exploits that use gadgets ending with indirect `jmp` or `call` instructions. More importantly, though, the significant runtime overhead imposed by the additional instrumentation instructions and the DBI framework itself limit their practical applicability.

ROPGuard [Fratric, 2012] is based on the observation that a ROP exploit will eventually invoke critical API functions, and performs various checks before such a function is called. These include checking whether `esp` is within the proper stack boundaries, whether a proper return address is present at the top of the stack, the consistency of stack frames, and other function-specific attributes. Although ROPGuard focuses only on non-JOP code, and some of its checks can result in false positives or can be easily evaded [Rosenberg, 2011; Portnoy, 2013], they are effective against current in-the-wild exploits, and some have been integrated in EMET [Microsoft, a].

ROPecker [Cheng *et al.*, 2014] detects the execution of ROP code by checking for chains of gadgets at runtime. Using the Last Branch Record feature of recent Intel CPUs, ROPecker gets access to the last 16 executed indirect branches at every checkpoint. Checkpoints are triggered whenever the execution leaves the last N pages of executable code, using a sliding window. The rationale behind this design decision is that a certain size of executable code is needed to construct useful ROP payloads [Schwartz *et al.*, 2011]. Evaluation results using the SPEC benchmarks showed a runtime performance overhead of only 2.6% both for page window sizes of 2 and 4. However, it is unclear whether larger and more complex, event-driven applications like web browsers have the same performance degradation behaviour.

Another set of defences uses the Branch trace storage (BTS) to record the execution of a running program. BTS is a debugging mechanism that enables the recording of all branch instructions in a user-defined memory area. However, the overhead due to the significant number of memory accesses, combined with the overall slower operation of the processor due to the special debug mode in which it enters when BTS is enabled, result to slowdowns typically in the range of 20–40× [Soffa *et al.*, 2011]. Consequently, systems that use BTS and similar mechanisms for control flow integrity [Xia *et al.*, 2012; Yuan *et al.*, 2011] or execution recording [Vasudevan *et al.*, 2011] suffer from significant runtime overheads.

A recent technique against kernel-level ROP uses the processor's performance counters to raise an interrupt after a number of mispredicted `ret` instructions, an indication of possible ROP code execution [Wicherski, 2013]. To rule out mispredicitons caused by legitimate code, upon an interrupt, the LBR stack is used to check whether the targets of the previously executed `ret` instructions are preceded by a `call` instruction. The use of JOP or call-preceded gadgets, however, can circumvent this protection.

Branch regulation [Kayaalp *et al.*, 2012] is a proposal for extending current processor architectures with a protection mechanism against ROP attacks. Besides maintaining a secondary call stack, the technique restricts the allowed targets of indirect `jmp` instructions to locations within the same function, or to the entry point of any other function, and only the latter for `call` instructions. Besides being quite restrictive for many legitimate programs, this approach requires protected binaries to go through a static binary instrumentation phase for annotating function boundaries, a process that requires precise code disassembly.

Chapter 3

In-Place Code Randomization

Starting with the goal of a practical mitigation against the recent spate of ROP attacks, in this chapter we present a novel code randomization method that can harden third-party applications against return-oriented programming. Our approach is based on narrow-scope modifications in the code segments of executables using an array of code transformation techniques, to which we collectively refer as *in-place code randomization.* These transformations are applied statically, in a conservative manner, and modify only the code that can be safely extracted from compiled binaries, without relying on symbolic debugging information. By preserving the length of instructions and basic blocks, these modifications do not break the semantics of the code, and enable the randomization of stripped binaries even without complete disassembly coverage. The goal of this randomization process is to eliminate or probabilistically modify as many of the gadgets that are available in the address space of a vulnerable process as possible. Since ROP code relies on the correct execution of all chained gadgets, altering the outcome of even a few of them will likely render the ROP code ineffective.

Currently, our design includes three code transformations that are applicable in-place. These are: (i) atomic instruction substitution, (ii) instruction reordering, and (iii) register reassignment. Although all of them seem applicable on most of the modern architecures (e.g., x86, ARM, MIPS), we developed a prototype implementation, named *Orp*, for the x86 32-bit architecture — source code is available at `http://nsl.cs.columbia.edu/projects/orp`. Our evaluation using real-world ROP exploits against widely used applications, such as Adobe Reader, shows the effectiveness and practicality of our approach, as in all cases the randomized versions of the applications rendered the exploits non-functional. When aiming to circumvent the applied code randomization, Q [Schwartz *et al.*, 2011] and Mona [Corelan Team, b], two automated ROP payload construction tools, were unable to generate functional exploit code by relying solely on any remaining non-randomized gadgets.

3.1 Approach

Our approach is based on the randomization of the code sections of binary executable files that are part of third-party applications, using an array of binary code transformation techniques. The objective of this randomization process is to break the code semantics of the gadgets that are present in the executable memory segments of a running process, without affecting the semantics of the actual program code.

The execution of a gadget has a certain set of consequences to the CPU and memory state of the exploited process. The attacker chooses how to link the different gadgets together based on which registers, flags, or memory locations each gadget modifies, and in what way. Consequently, the execution of a subsequent gadget depends on the outcome of all previously executed gadgets. Even if the execution of a single gadget has a different outcome than the one anticipated by the attacker, then this will affect the execution of all subsequent gadgets, and it is likely that the logic of the malicious return-oriented code will be severely impacted.

3.1.1 Why In-Place?

The concept of software diversification [Cohen, 1993] is the basis for a wide range of protections against the exploitation of memory corruption vulnerabilities. Besides address space layout randomization [Miller *et al.*, 2011], many techniques focus on the internal randomization of the code segments of executable, and can be combined with ASLR to increase process diversity [Forrest *et al.*, 1997]. Metamorphic transformations [Ször, 2005] can shift gadgets from their original offsets and alter many of their instructions, rendering them unusable. Another simpler and probably more effective approach is to rearrange existing blocks of code either at the function level [Bhatkar *et al.*, 2003; Bhatkar *et al.*, 2005; Kil *et al.*, 2006; Microsoft, b], or with finer granularity, at the basic block level [Google, 2011; Microsoft, c]. If all blocks of code are reordered so that no one resides at its original location, then all the offsets of the gadgets that the attacker would assume to be present in the code sections of the process will now correspond to completely different code.

These transformations require a precise view of all the code and data objects contained in the executable sections of a PE file, including their cross-references, as existing code needs to be shifted or moved. Due to computed jumps and intermixed data [Kruegel *et al.*, 2004], complete disassembly coverage is possible only if the binary contains relocation and symbolic debugging information (e.g., PDB files) [Smithson *et al.*, 2010; Kil *et al.*, 2006; Saxena *et al.*, 2008]. Unfortunately, debugging information is typically stripped from release builds for compactness and intellectual property protection.

For Windows software, in particular, PE files (both DLL and EXE) usually do retain relocation information even if no debugging information has been retained [Skape, 2007]. The loader needs this information in case a DLL must be loaded at an address other than its preferred base address, e.g., because another library has already been mapped to that location. or for ASLR. In contrast to Linux shared libraries and PIC executables, which contain position-independent code, Windows binaries contain absolute addresses, e.g., as immediate instruction operands or initialized data pointers, that are valid only if the executable has been loaded at its preferred base address. The `.reloc` section of PE files contains a list of offsets relatively to each PE section that correspond to all absolute addresses at which a delta value needs to be added in case the actual load address is different [Pietrek, 2002].

Relocation information *alone*, however, does not suffice for extracting a complete view of the code within the executable sections of a PE file [Google, 2011; Smithson *et al.*, 2010]. Without the symbolic debugging information contained in PDB files, although the location of objects that are reached *only* via indirect jumps *can* be extracted from relocation information, their actual type—code or data—still remains unknown. In some cases, the actual type of these objects could be inferred using heuristics based on constant propagation, but such methods are usually prone to misidentifications of data as code and vice versa. Even a slight shift or size increase of a single object within a PE section will incur cascading shifts to its following objects. Typically, an unidentified object that actually contains code will include PC-relative branches to other code objects. In the absence of the debugging information contained in PDB files, moving such an unidentified code block (or any of its relatively referenced objects) without fixing the displacements of all its relative branch instructions that reference other objects, will result to incorrect code.

Given the above constraints, we choose to use only binary code transformations that do not alter the size and location of code and data objects within the executable, allowing the randomization of third-party PE files *without* symbolic debugging information. Although this restriction does not allow us to apply extensive code transformations like basic block reordering or metamorphism, we can still achieve partial code randomization using narrow-scope modifications that can be *safely* applied even without complete disassembly coverage. This can be achieved through slight, in-place code modifications to the correctly identified parts of the code, that do not change the overall structure of basic blocks or functions, but which are enough to alter the outcome of short instruction sequences that can be used as gadgets.

3.1.2 Code Extraction and Modification

Although completely accurate disassembly of stripped x86 binaries is not possible, state-of-the-art disassemblers achieve decent coverage for code generated by the most commonly used compilers, using a combination of different disassembly algorithms [Kruegel *et al.*, 2004], the identification of specific code constructs [Guilfanov, 2008b], and simple data flow analysis [Guilfanov, 2008a]. For our prototype implementation, we use IDA Pro [Hex-Rays,] to extract the code and identify the functions of PE executables. IDA Pro is effective in the identification of function boundaries, even for functions with non-contiguous code and extensive use of basic block sharing [Hu *et al.*, 2009], and also takes advantage of the relocation information present in Windows DLLs.

Typically, however, without the symbolic information of PDB files, a fraction of the functions in a PE executable are not identified, and parts of code remain undiscovered. Our code transformations are applied conservatively, only on parts of the code for which we can be confident that have been accurately disassembled. For instance, IDA Pro speculatively disassembles code blocks that are reached only through computed jumps, taking advantage of the relocation information contained in PE files. However, we do not enable such heuristic code extraction methods in order to avoid any disastrous modifications due to potentially misidentified code. In practice, for the code generated by most compilers, relocation information also ensures that the correctly identified basic blocks have no entry point other than their first instruction. Similarly, some transformations that rely on the proper identification of functions are applied only on the code of correctly recognized functions. Our implementation is separate from the actual code extraction framework used, which means that IDA Pro can be replaced or assisted by alternative code extraction approaches [Nanda *et al.*, 2006; Smithson *et al.*, 2010; Harris and Miller, 2005], providing better disassembly coverage.

After code extraction, disassembled instructions are first converted to our own internal representation, which holds additional information such as any implicitly used registers, and the registers and flags read or written by the instruction. For correctness, we also track the use of general purpose registers even in floating point, MMX, and SSE instructions. Although these type of instructions have their own set of registers, they do use general purpose registers for memory references (e.g., as the `fmul` instruction in Fig. 3.1). We then proceed and apply the in-place code transformations discussed in the following section. These are applied only on the parts of the executable segments that contain (intended or unintended [Shacham, 2007]) instruction sequences that can be used as gadgets. As a result of some of the transformations, instructions may be moved from their original locations within the same basic block. In these cases, for instructions that contain an absolute address in some of their operands, the corresponding entries in the `.reloc` sections of the randomized PE file are updated with the new offsets where these absolute addresses are now located.

Our prototype implementation processes each PE file individually, and generates multiple randomized copies that can then replace the original. Given the complexity of the analysis required for generating a set of randomized instances of an input file (in the order of a few minutes on average for the PEs used in our tests), this allows the off-line generation of a pool of randomized PE files for a given application. Note that for most of the tested Windows applications, only some of the DLLs need to be randomized, as the rest are usually ASLR-enabled (although they can also be randomized for increased protection). In a production deployment, a system service or a modified loader can then pick a different randomized version of the required PEs each time the application is launched, following the same way of operation as tools like EMET [Microsoft, a].

3.2 In-Place Code Transformations

In this section we present in detail the different code transformations used for in-place code randomization. Although some of the transformations such as instruction reordering and register reassignment are also used by compilers and polymorphic code engines for code optimization [Aho *et al.*, 2006] and obfuscation [Ször, 2005], applying them at the binary level—without having access to the higher-level structural and semantic information available in these settings—poses significant challenges.

3.2.1 Atomic Instruction Substitution

One of the basic concepts of code obfuscation and metamorphism [Ször, 2005] is that the exact same computation can be achieved using a countless number of different instruction combinations. When applied for code randomization, substituting the instructions of a gadget with a functionally-equivalent—but different—sequence of instructions would not affect any ROP code that uses that gadget, since its outcome would be the same. However, by modifying the instructions of the original program code, this transformation in essence modifies certain bytes in the code image of the program, and consequently, can drastically alter the structure of non-intended instruction sequences that overlap with the substituted instructions.

Many of the gadgets used in ROP code consist of unaligned instructions that have not been emitted by the compiler, but which happen to be present in the code image of the process due to the density and variable-length nature of the x86 instruction set. In the example of Fig. 3.1(a), the actual code generated by the compiler consists of the instructions `mov; cmp; lea;` starting at byte `B0`.[1] However, when disassembling from the next byte,

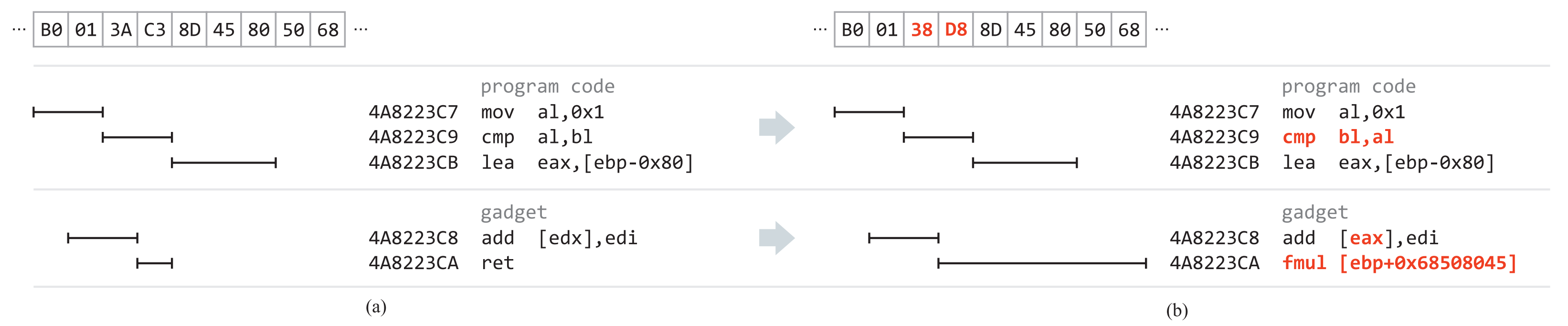
[1] The code of all examples comes from icucnv36.dll, included in Adobe Reader v9.3.4.

a useful non-intended gadget ending with `ret` is found.

Compiled code is highly optimized, and thus the replacement of even a single instruction in the original program code usually requires either a longer instruction, or a combination of more than one instruction, for achieving the same purpose. Given that our aim is to randomize the code of stripped binaries, even a slight increase in the size of a basic block is not possible, which makes the most commonly used instruction substitution techniques unsuitable for our purpose.

In certain cases though, it is possible to replace an instruction with a single, functionally-equivalent instruction of the *same* length, thanks to the flexibility offered by the extensive x86 instruction set. Besides obvious candidates based on replacing addition with negative subtraction and inversely, there are also some instructions that come in different forms, with different opcodes, depending on the supported operand types. For example, `add r/m32,r32` stores the result of the addition in a register *or* memory operand (r/m32), while `add r32,r/m32` stores the result in a register (r32). Although these two forms have different opcodes, the two instructions are equivalent when both operands happen to be registers. Many arithmetic and logical instructions have such dual equivalent forms, while in some cases there can be up to five equivalent instructions (e.g., `test r/m8,r8`, `or r/m8,r8`, `or r8, r/m8`, `and r/m8,r8`, `and r8,r/m8`, affect the flags of the EFLAGS register in the same way when both operands are the *same* register). In our prototype implementation we use the sets of equivalent instructions used in Hydan [El-Khalil and Keromytis, 2004], a tool for hiding information in x86 executables, with the addition of one more set that includes the equivalent versions of the `xchg` instruction.

As shown in Fig. 3.1(b), both operands of the `cmp` instruction are registers, and thus it can be replaced by its equivalent form, which has different opcode and ModR/M bytes [Intel, 2014]. Although the actual program code does not change, the `ret` instruction that was "included" in the original `cmp` instruction has now disappeared, rendering the gadget unusable. In this case, the transformation completely *eliminates* the gadget, and thus will be applied in all instances of the randomized binary. In contrast, when a substitution does not affect the gadget's final indirect jump, then it is applied probabilistically.



```
… B0 01 3A C3 8D 45 80 50 68 …

program code
4A8223C7  mov   al,0x1
4A8223C9  cmp   al,bl
4A8223CB  lea   eax,[ebp-0x80]

gadget
4A8223C8  add   [edx],edi
4A8223CA  ret
```

(a)

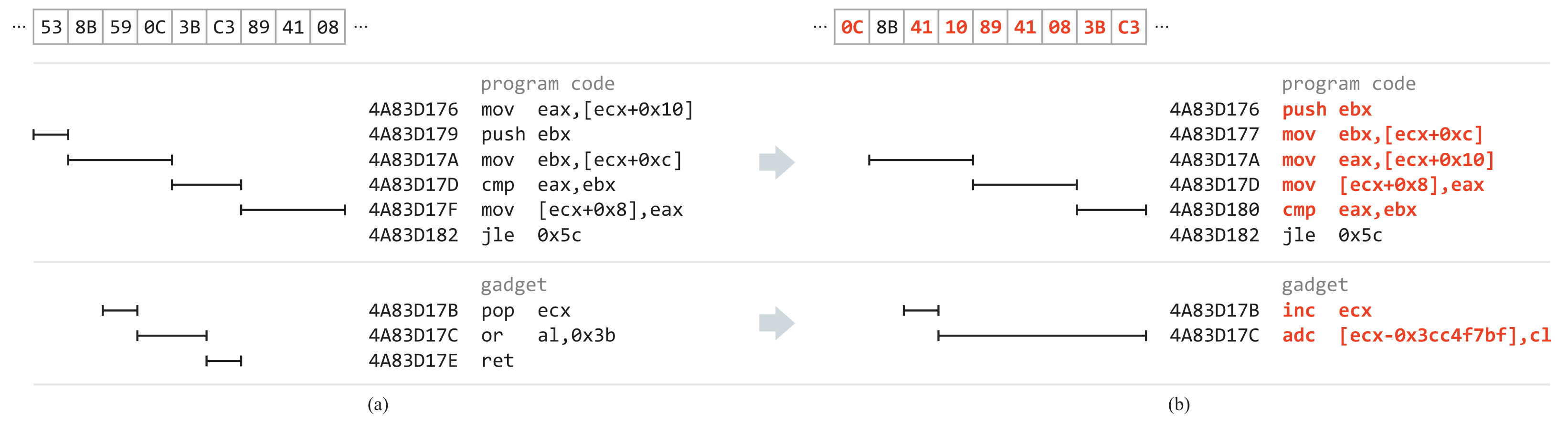
```
… B0 01 38 D8 8D 45 80 50 68 …

program code
4A8223C7  mov   al,0x1
4A8223C9  cmp   bl,al
4A8223CB  lea   eax,[ebp-0x80]

gadget
4A8223C8  add   [eax],edi
4A8223CA  fmul  [ebp+0x68508045]
```

(b)

Figure 3.1: Example of atomic instruction substitution. The equivalent, but different form of the cmp instruction does not change the original program code (a), but renders the non-intended gadget unusable (b).



```
… 53 8B 59 0C 3B C3 89 41 08 …

program code
4A83D176  mov   eax,[ecx+0x10]
4A83D179  push  ebx
4A83D17A  mov   ebx,[ecx+0xc]
4A83D17D  cmp   eax,ebx
4A83D17F  mov   [ecx+0x8],eax
4A83D182  jle   0x5c

gadget
4A83D17B  pop   ecx
4A83D17C  or    al,0x3b
4A83D17E  ret
```

(a)

```
… 0C 8B 41 10 89 41 08 3B C3 …

program code
4A83D176  push  ebx
4A83D177  mov   ebx,[ecx+0xc]
4A83D17A  mov   eax,[ecx+0x10]
4A83D17D  mov   [ecx+0x8],eax
4A83D180  cmp   eax,ebx
4A83D182  jle   0x5c

gadget
4A83D17B  inc   ecx
4A83D17C  adc   [ecx-0x3cc4f7bf],cl
```

(b)

Figure 3.2: Example of how intra basic block instruction reordering can affect a non-intended gadget.

3.2.2 Instruction Reordering

In certain cases, it is possible to reorder the instructions of small self-contained code fragments without affecting the correct operation of the program. This transformation can significantly impact the structure of non-intended gadgets, but can also break the attacker's assumptions about gadgets that are part of the actual machine code.

3.2.2.1 Intra Basic Block Reordering

The actual instruction scheduling chosen during the code generation phase of a compiler depends on many factors, including the cost of instructions in cycles, and the applied code optimization techniques [Aho *et al.*, 2006]. Consequently, the code of a basic block is often just one among several possible instruction orderings that are all equivalent in terms of correctness. Based on this observation, we can partially modify the code within a basic block by reordering some of its instructions according to an alternative instruction scheduling.

The basis for deriving an alternative instruction scheduling is to determine the ordering relationships among the instructions, which must always be satisfied to maintain code correctness. The *dependence graph* of a basic block represents the instruction interdependencies that constrain the possible instruction schedules [Muchnick, 1997]. Since a basic block contains straight-line code, its dependence graph is a directed acyclic graph with machine instructions as vertices, and dependencies between instructions as edges. We apply dependence analysis on the code of disassembled basic blocks to build their dependence graph using an adaptation of a standard dependence DAG construction algorithm [Muchnick, 1997, Fig. 9.6] for machine code. Applying dependence analysis directly on machine code requires a careful treatment of the dependencies between x86 instructions. Compared to the analysis of code expressed in an intermediate representation form, this includes the identification of data dependencies not only between register and memory operands, but also between CPU flags and implicitly used registers and memory locations.

For each instruction i, we derive the sets $use[i]$ and $def[i]$ with the registers used and defined by the instruction. Besides register operands and registers used as part of effective address computations, this includes any implicitly used registers. For example, the *use* and *def* sets for `pop eax` are $\{esp\}$ and $\{eax, esp\}$, while for `rep stosb`[2] are $\{ecx, eax, edi\}$ and $\{ecx, edi\}$, respectively. We initially assume that all instructions in the basic block depend on each other, and then check each pair for read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies. For example, i_1 and i_2 have a RAW dependency if any of the following conditions is true: i) $def[i_1] \cap use[i_2] \neq \emptyset$, ii) the destination operand of i_1 and the source operand of i_2 are both a memory location, iii) i_1 writes at least one flag read by i_2.

Note that condition ii) is quite conservative, given that i_2 will actually depend on i_1 only if i_2 reads the *same* memory location written by i_1. However, unless both memory operands use absolute addresses, it is hard to determine statically if the two effective addresses point to the same memory location. In our future work, we plan to use simple data flow analysis to relax this condition. Besides instructions with memory operands, this condition should also be checked for instructions with implicitly accessed memory locations, e.g., `push` and `pop`. The conditions for WAR and WAW dependencies are analogous. If no conflict is found between two instructions, then there is no constraint in their execution order.

Figure 3.2(a) shows the code of a basic block that contains a non-intended gadget, and Fig. 3.3 its corresponding dependence DAG. Instructions not connected via a direct edge are independent, and have no constraint in their relative execution order. Given the dependence DAG of a basic block, the possible orderings of its instructions correspond to the different topological sorting arrangements of the graph [Varol and Rotem, 1981]. Fig. 3.2(b) shows one of the possible alternative orderings of the original code. The locations of all but one of the instructions and the values of all but one of the bytes have changed, eliminating the non-intended gadget contained in the original code. Although a new gadget has appeared a few bytes further into the block, (ending again with a `ret` instruction at byte `C3`), an attacker cannot depend on it since alternative orderings will shift it to other locations, and some of its internal instructions will always change (e.g., in this example, the useful `pop ecx` is gone). In fact, the `ret` instruction can be eliminated altogether using atomic

[2] `stosb` (Store Byte to String) copies the least significant byte from the `eax` register to the memory location pointed by the `edi` register and increments `edi`'s value by one. The `rep` prefix repeats this instruction until `ecx`'s value reaches zero, while decreasing it after each repetition.

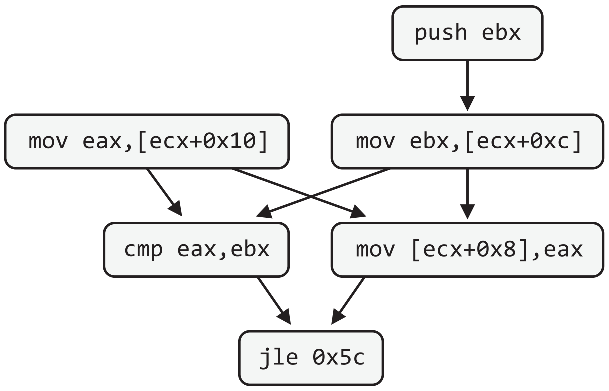


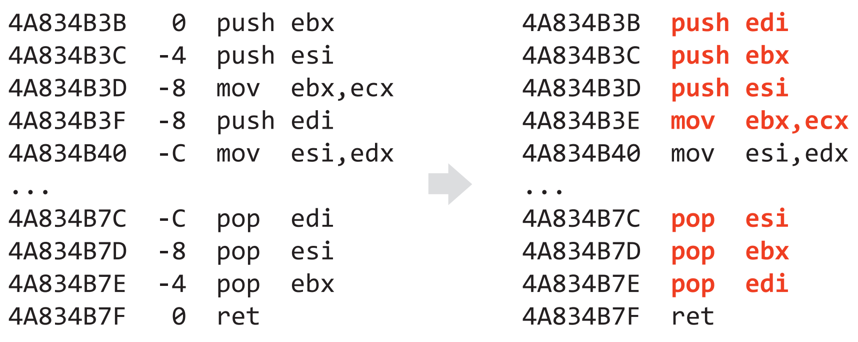
Figure 3.3: Dependence graph for the code of Fig. 3.2.

instruction substitution.

An underlying assumption we make here is that basic block boundaries will not change at runtime. If a computed control transfer instruction targets a basic block instruction other than its first, then reordering may break the semantics of the code. Although this may seem restrictive, we note that throughout our evaluation we did not encounter any such case. For compiler-generated code, IDA Pro is able to compute all jump targets even for computed jumps based on the PE relocation information. In the most conservative case, users may choose to disable instruction reordering and still benefit from the randomization of the other techniques—Section 3.3 includes results for each technique individually.

3.2.2.2 Reordering of Register Preservation Code

The calling convention followed by the majority of compilers for Windows on x86 architectures, similarly to Linux, specifies that the `ebx`, `esi`, `edi`, and `ebp` registers are callee-saved [Fog,]. The remaining general purpose registers, known as scratch or volatile registers, are free for use by the callee without restrictions. Typically, a function that needs to use more than the available scratch registers, preserves any non-volatile registers before modifying them by storing their values on the stack. This is usually done at the function prologue through a series of `push` instructions, as in the example of Fig. 3.4(a), which shows the very first and last instructions of a function. At the function epilogue, a corresponding series of `pop` instructions restores the saved values from the stack, right before returning to the



```
4A834B3B   0  push ebx
4A834B3C  -4  push esi
4A834B3D  -8  mov  ebx,ecx
4A834B3F  -8  push edi
4A834B40  -C  mov  esi,edx
...
4A834B7C  -C  pop  edi
4A834B7D  -8  pop  esi
4A834B7E  -4  pop  ebx
4A834B7F   0  ret
```

(a)

```
4A834B3B  push edi
4A834B3C  push ebx
4A834B3D  push esi
4A834B3E  mov  ebx,ecx
4A834B40  mov  esi,edx
...
4A834B7C  pop  esi
4A834B7D  pop  ebx
4A834B7E  pop  edi
4A834B7F  ret
```

(b)

Figure 3.4: Example of register preservation code reordering.

caller. Sequences that contain `pop` instructions followed by `ret` are among the most widely used gadgets found in ROP exploits, since they allow the attacker to load registers with values that are supplied as part of the injected payload [Skape and Skywing, 2005]. The order of the `pop` instructions is crucial for initializing each register with the appropriate value.

As seen in the function prologue, the compiler stores the values of the callee-saved registers in arbitrary order, and sometimes the relevant `push` instructions are interleaved with instructions that use previously-preserved registers. At the function epilogue, the saved values are `pop`'ed from the stack in *reverse* order, so that they end up to the proper register. Consequently, as long as the saved values are restored in the right order, their actual order on the stack is irrelevant. Based on this observation, we can randomize the order of the `push` and `pop` instructions of register preservation code by maintaining the first-in-last-out order of the stored values, as shown in Fig. 3.4(b). In this example, there are six possible orderings of the three `pop` instructions, which means that any assumption that the attacker may make about which registers will hold the two supplied values, will be correct with a probability of one in six (or one in three, if only one register needs to be initialized). In case only two registers are preserved, there are two possible orderings, allowing the gadget to operate correctly half of the time.

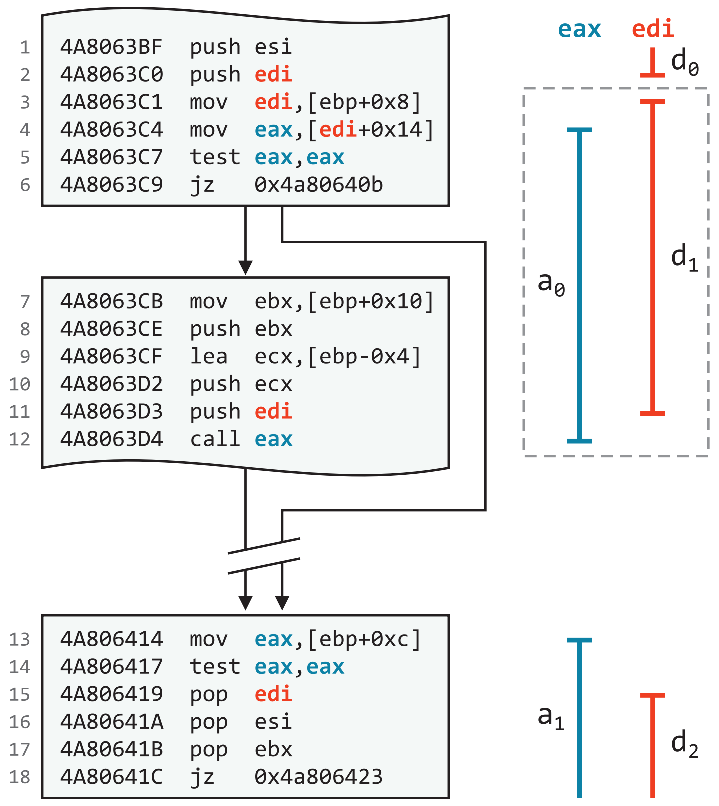
This transformation is applied conservatively, only to functions with accurately disassembled prologue and epilogue code. To make sure that we properly match the `push` and `pop` instructions that preserve a given register, we monitor the stack pointer delta throughout the whole function, as shown in the second column of Fig. 3.4(a). If the deltas at the prologue and epilogue do not match, e.g., due to call sites with unknown calling conventions throughout the function, or indirect manipulation of the stack pointer, then no randomization is applied. As shown in Fig. 3.4(b), any non-preservation instructions in the function prologue are reordered along with the `push` instructions by maintaining any interdependencies, as discussed in the previous section. For functions with multiple exit points, the preservation code at all epilogues should match the function's prologue. Note that there can be multiple `push` and `pop` pairs for the same register, in case the register is preserved only throughout some of the execution paths of a function.

3.2.3 Register Reassignment

Although the program points at which a certain variable should be stored in a register or spilled into memory are chosen by the compiler using sophisticated allocation algorithms, the actual name of the general purpose register that will hold a particular variable is mostly an arbitrary choice. Based on this observation, we can reassign the names of the register operands in the existing code according to a different—but equivalent—register assignment, without affecting the semantics of the original code. When considering each gadget as an autonomous code sequence, this transformation can alter the outcome of many gadgets, which will now read or modify different registers than those assumed by the attacker.

Due to the much higher cost of memory accesses compared to register accesses, compilers strive to map as many variables as possible to the available registers. Consequently, at any point in a large program, multiple registers are usually in use, or *live* at the same time. Given the control flow graph (CFG) of a compiled program, a register r is *live* at a program point p iff there is a path from p to a use of r that does not go through a definition of r. The *live range* of r is defined as the set of program points where r is live, and can be represented as a subgraph of the CFG [Bouchez, 2009]. Since the same register can hold different variables at different points in the program, a register can have multiple disjoint live regions in the same CFG.

For each correctly identified function, we compute the live ranges of all registers used in its body by performing liveness analysis [Aho *et al.*, 2006] directly on the machine code.



```
 1  4A8063BF  push esi
 2  4A8063C0  push edi
 3  4A8063C1  mov  edi,[ebp+0x8]
 4  4A8063C4  mov  eax,[edi+0x14]
 5  4A8063C7  test eax,eax
 6  4A8063C9  jz   0x4a80640b

 7  4A8063CB  mov  ebx,[ebp+0x10]
 8  4A8063CE  push ebx
 9  4A8063CF  lea  ecx,[ebp-0x4]
10  4A8063D2  push ecx
11  4A8063D3  push edi
12  4A8063D4  call eax

13  4A806414  mov  eax,[ebp+0xc]
14  4A806417  test eax,eax
15  4A806419  pop  edi
16  4A80641A  pop  esi
17  4A80641B  pop  ebx
18  4A80641C  jz   0x4a806423
```

Figure 3.5: The live ranges of `eax` and `edi` in part of a function. The two registers can be swapped in all instructions throughout their parallel, self-contained regions a_0 and d_1 (lines 3–12).

Given the CFG of the function and the sets $use[i]$ and $def[i]$ for each instruction i, we derive the sets $in[i]$ and $out[i]$ with the registers that are *live-in* and *live-out* at each instruction. For this purpose, we use a modified version of a standard live-variable analysis algorithm [Aho *et al.*, 2006, Fig. 9.16] that computes the *in* and *out* sets at the instruction level, instead of the basic block level. The algorithm computes the two sets by iteratively reaching a fixed point for the following data-flow equations: $in[i] = use[i] \cup (out[i] - def[i])$ and $out[i] = \bigcup\{in[s] : s \in succ[i]\}$, were $succ[i]$ is the set of all possible successors of instruction i.

Figure 3.5 shows part of the CFG of a function and the corresponding live ranges for `eax` and `edi`. Initially, we assume that all registers are live, since some of them may hold values that have been set by the caller. In this example, `edi` is live when entering the function, and the `push` instruction at line 2 stores (uses) its current value on the stack. The following `mov` instruction initializes (defines) `edi`, ending its previous live range (d_0). Note that although a live range is a sub-graph of the CFG, we illustrate and refer to the different live ranges as linear regions for the sake of convenience.

The next definition of `edi` is at line 15, which means that the last use of its previous value at line 11 also ends its previous live region d_1. Region d_1 is a *self-contained* region, within which we can be confident that `edi` holds the same variable. The `eax` register also has a self-contained live region (a_0) that runs in parallel with d_1. Conceptually, the two live ranges can be extended to share the same boundaries. Therefore, the two registers can be swapped across all the instructions located within the boundaries of the two regions, without altering the semantics of the code.

The `call eax` instruction at line 12 can be conveniently used by an attacker for calling a library function or another gadget. By reassigning `eax` and `edi` across their parallel live regions, any ROP code that would depend on `eax` for transferring control to the next piece of code, will now jump to an incorrect memory location, and probably crash. For code fragments with just two parallel live regions, an attacker can guess the right register half of the times. In many cases though, there are three or more general purpose registers with parallel live regions, or other available registers that are live before or after another register's live region, allowing for a higher number of possible register assignments.

The registers used in the original code can be reassigned by modifying the ModR/M and sometimes the SIB byte of the relevant instructions. As in previous code transformations, besides altering the operands of instructions in the existing code, these modifications can also affect overlapping instructions that may be part of non-intended gadgets. Note that implicitly used registers in certain instructions cannot be replaced. For example, the one-byte "move data from string to string" instruction (`movs`) always uses `esi` and `edi` as its source and destination operands, and there is no other one-byte instruction for achieving the same operation using a different set of registers [Intel, 2014]. Consequently, if such an instruction is part of the live region of one of its implicitly used registers, then this register cannot be reassigned throughout that region. For the same reason, we exclude `esp` from liveness analysis. Finally, although calling conventions are followed for most of the functions, this is not always the case, as compilers are free to use any custom calling convention for private or static functions. Most of these cases are conservatively covered through a bottom-up call analysis that discovers custom register arguments and return value registers.

First, all the external function definitions found in the import table of the DLL are marked as level-0 functions. IDA Pro can effectively distinguish between different calling conventions that these external functions may follow, and reports their declaration in the C language. Thus, in most cases, the register arguments and the return value register (if any) for each of the level-0 functions are known. For any `call` instruction to a level-0 function, its register arguments are added to `call`'s set of implicitly read registers, and its return value registers are added to `call`'s set of implicitly written registers.

In the next phase, level-1 functions are identified as the set of functions that call only level-0 functions or no other function. Any registers read by a level-1 function, without prior writing them, are marked as its register arguments. Similarly, any registers written and not read before a return instruction are marked as return value registers. Again, the sets of implicitly read and written register of all the `call` instructions to level-1 functions are updated accordingly. Similarly, level-2 functions are the ones that call level-1 or level-0 functions, or no other function, and so on. The same process is repeated until no more function levels can be computed. The intuition behind this approach is that private functions, which may use non-standard calling conventions, are called by other functions in the same DLL and, in most cases, not through computed call instructions.

3.3 Experimental Evaluation

In this section, we experimentally evaluate in-place code randomization along three axis. First, we perform an analysis of the randomization in terms of coverage of the executable code, impact to the affected gadgets and randomization entropy, using a large dataset of more that five thousands DLLs. Second, we verify the correctness and performance of the technique using part of the Wine test suite. And, third, we evaluate its effectiveness using a set of real-world exploits and two automated ROP construction toolkits.

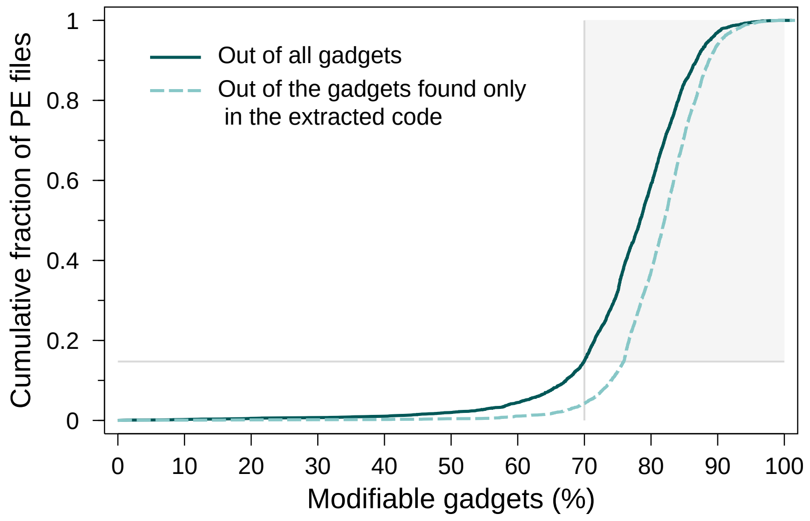


Figure 3.6: Percentage of modifiable gadgets for a set of 5,235 PE files. Indicatively, for the upper 85% of the files, more than 70% of *all* gadgets in the executable segments of each PE file can be modified (shaded area).

3.3.1 Randomization Analysis

3.3.1.1 Coverage

A crucial aspect for the effectiveness of in-place code randomization is the randomization coverage in terms of what percentage of the gadgets found in an executable can be safely randomized. A gadget may remain intact for one of the following reasons: i) it is part of data embedded in a code segment, ii) it is part of code that could not be disassembled, or iii) it is not affected by any of our transformations. In this section, we explore the randomization coverage of our prototype implementation using a large data set of 5,235 PE files (both DLL and EXE), detailed in Table 3.1.

We consider as a gadget [Shacham, 2007] any intended or unintended instruction sequence that ends with an indirect control transfer instruction, and which does not contain i) a privileged or invalid instruction (can occur in non-intended instruction sequences), and ii) a control transfer instruction other than its final one, with the exception of indirect `call` (can be used in the middle of a gadget for calling a library function). We assume a maximum gadget length of five instructions, which is typical for existing ROP code imple-

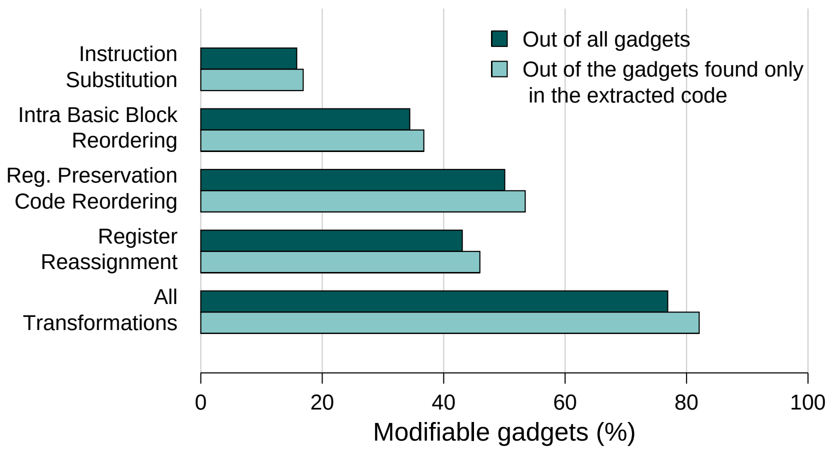


Figure 3.7: Percentage of modifiable gadgets according to the different code transformations.

Table 3.1: Modifiable (eliminated vs. broken) gadgets for a collection of various PE files.

Software	PE Files	Code (MB)	Total	Modifiable (%)	Eliminated (%)	Broken (%)
Adobe Reader 9	43	6.7	1,250K	943K (75.4)	108K (8.7)	834K (66.7)
Firefox 4	28	3.5	458K	381K (83.0)	56K (12.4)	324K (70.6)
iTunes 10	75	3.7	396K	293K (74.0)	31K (8.0)	261K (66.0)
Windows XP SP3	1,698	134.4	8,305K	6,452K (77.7)	770K (9.3)	5,682K (68.4)
Windows 7 SP1	3,391	324.8	16,951K	12,970K (76.5)	1,637K (9.7)	11,333K (66.8)
Total	5,235	473.1	27,362K	21,041K (**76.9**)	2,604K (**9.5**)	18,436K (**67.4**)

mentations [Shacham, 2007; Checkoway *et al.*, 2010]. For larger gadgets, it is possible that the modified part of the gadget may be irrelevant for the purpose of the attacker. For example, if only the first instruction of the gadget `inc eax; pop ebx; ret;` is randomized, this will not affect any ROP code that either does not rely on the value of `eax` at that point, or uses the shorter gadget `pop ebx; ret;` directly. For this reason, we consider *all* different subsequences with length between two to five instructions as separate gadgets.

Figure 3.6 shows the percentage of modifiable gadgets out of *all* gadgets found in the executable sections of each PE file (solid line), as a cumulative fraction of all PE files in the data set. In about 85% of the PE files, more that 70% of the gadgets can be randomized by our code transformations. Many of the unmodified gadgets are located in parts of code that have not been extracted by IDA Pro, and which consequently will never be affected

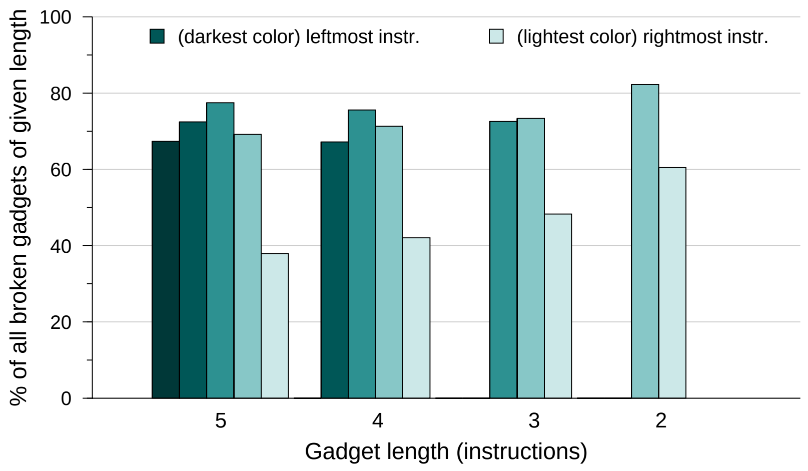


Figure 3.8: Impact of code randomization on the broken gadgets' instructions according to their location in the gadget. The order of the bars corresponds to the order of the instructions in the gadget. Indicatively, the first (leftmost) instruction of two-instruction gadgets is altered in more than 80% of all broken two-instruction gadgets.

by our transformations. When considering only the gadgets that are contained within the disassembled code regions on which code randomization can be applied, the percentage of affected gadgets slightly increases (dashed line). Given that we do not take into account code blocks that have been identified by IDA Pro using speculative methods, this shows that the use of a more sophisticated code extraction mechanism will increase the number of gadgets that can be modified. Figure 3.7 shows the total percentage of gadgets modified by each code transformation technique for the same data set. Note that a gadget can be modified by more than one technique. Overall, the total percentage of modifiable gadgets across all PE files is about 76.9%, as shown in Table 3.1.

3.3.1.2 Impact

We identify two qualitatively different ways in which a code transformation can impact a gadget. As discussed in Sec. 3.2.1, a gadget can be *eliminated*, if any of the applied transformations removes completely its final control transfer instruction. If the final control transfer instruction remains intact, a gadget can then be *broken*, if at least one of its internal instructions is altered, and the CPU and memory state after its execution is different than the original, i.e., the outcome of its computation is not the same. As shown in Table 3.1, in the average case, about 9.5% of *all* gadgets contained in a PE file can be rendered completely unusable. For a vulnerable application, this already removes about one in ten of the available gadgets for the construction of ROP code. Although the rest of the modifiable gadgets (67.4%) is not eliminated, they can be "broken" by probabilistically modifying one or more of their instructions.

In case some of the instructions in a broken gadget can never be altered, it is quite possible that part of its functionality will remain unaffected, and thus an attacker could still use it by relying only on its unmodifiable instructions. Especially for larger gadget sizes, if the possible modifications are clustered only around a certain part of the gadget, e.g., its first instructions, then an attacker could predictably rely on the rest of the gadget. We explore this issue by measuring the number of broken gadgets in which an instruction at a given position can be altered.

Figure 3.8 shows the impact of code randomization on a broken gadget's instructions, according to their location within the gadget. Each group of bars corresponds to a different gadget length, and in each group, the leftmost bar corresponds to the leftmost instruction of the gadget. For all sizes, the probability that an instruction at a given position will be affected is quite evenly distributed and remains beyond 60%, with the exception of the final (control transfer) instruction. This is expected, since most of the transformations cannot affect the final instruction of intended gadgets (e.g., `ret`). As we observe, the locations of the modified instructions in broken gadgets are almost equally unpredictable.

3.3.1.3 Entropy

Some of the code transformations can perturb a given instruction within a gadget only in a limited number of ways, while others can generate a larger number of permutations. For example, for instructions with only two equivalent forms, atomic instruction substitution can modify a particular location in a gadget only in one way, allowing for two possible states. On the other hand, intra basic block instruction reordering usually results to a large number of possible permutations, especially for larger basic blocks that contain many instructions with no interdependencies.

Usually though, a broken gadget can be modified at multiple locations, and the same location can be altered in multiple ways by more than one code transformations. Consequently, the number of possible randomized states in which a broken gadget can exist, or its randomization *entropy*, corresponds to the product of the number of permutations that each of the different transformations can generate for that gadget. In the worst case, a broken gadget can exist in two possible states: its original form, or its alternative after modification. For example, there are only two possible orderings for the `pop` instructions in an intended gadget of the form `pop reg; pop reg; ret;` given that no other transformation can alter it.

Figure 3.9 shows the number of possible randomized versions of each gadget (including its original), as a cumulative fraction of all broken gadgets. As seen in the lower left corner, a small amount of about 12% of the gadgets can be modified only in one way, and thus can exist in two possible states. However, the randomization entropy increases exponentially, and the upper 80% of the gadgets have four or more randomized states. As more of the different transformations are applied on the same gadget, the randomization entropy increases to thousands of possible modified states.

Although for a small amount of gadgets an attacker can have a 50% chance of guessing the actual behavior of a gadget, ROP code relies on a chain of many different gadgets to achieve its purpose (11–18 unique gadgets in the exploits we tested). Even if one of the gadgets behaves in a non-expected way, then the ROP code will fail. Given that code randomization typically breaks (or even eliminates) several of the gadgets used in a ROP exploit, the number of possible randomized states that can prevent the correct execution of the ROP code is usually very high, as demonstrated in Sec. 3.3.3.

3.3.2 Correctness and Performance

One of the basic principles of our approach is that the different in-place code randomization techniques should be applied cautiously, without breaking the semantics of the program. A straightforward way to verify the correctness of our code transformations is to apply them on existing code and compare the outcome before and after modification. Simply running a randomized version of a third-party application and verifying that it behaves in

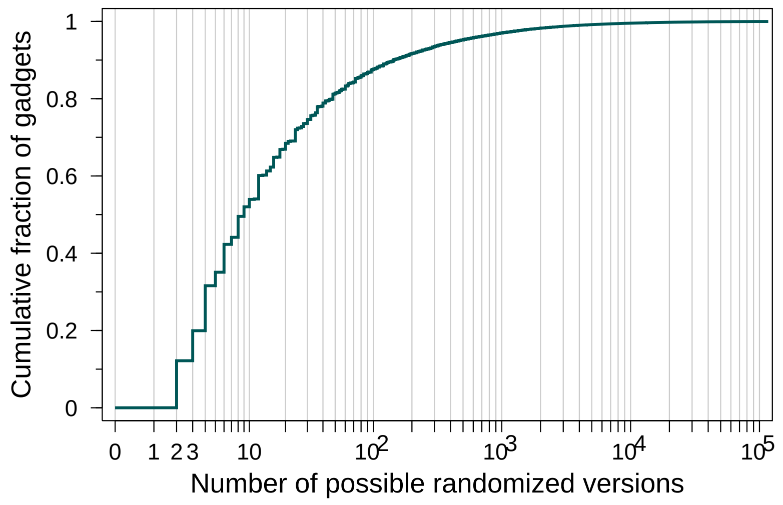


Figure 3.9: Randomization entropy for broken gadgets.

the expected way can provide a first indication. However, using this approach, it is hard to exercise a significant part of the code, and potentially incorrect modifications may go unnoticed.

For this purpose, we used the test suite of Wine [Wine,], a compatibility layer that allows Windows applications to run on Unix-like operating systems. Wine provides alternative implementations of the DLLs that comprise the Windows API, and comes with an extensive test suite that covers the implementations of most functions exported by the core Windows DLLs. Each function is executed multiple times using various inputs that test different conditions, and the outcome of each execution is compared against a known, expected result. We ported the test code for about one third of the 109 DLLs included in the test suite of Wine v1.2.2, and used it directly on the actual DLLs from a Windows 7 installation. Using multiple randomized versions of each tested DLL, we verified that in all runs, all tests completed successfully.

We took advantage of the extensive and diverse code execution coverage of this experiment to also evaluate the impact of in-place code randomization to the runtime performance of the modified code. Among the different code transformations, instruction reordering is the only one that could potentially introduce some non-negligible overhead, given that sometimes the chosen ordering may be sub-optimal. We measured the overall CPU user time

Table 3.2: ROP exploitsand generic ROP payloadstested on Windows 7 SP1.

ROP exploit / payload	non-ASLR DLLs: used for ROP	Total Gadgets	Modifiable (total %: Broken % Eliminated %)	Unique Gadgets Used: Modifiable (Br.,El.)	Combin-ations
Adobe Reader	3: 1	36,760	28,637 (77.9: 70.1 7.8)	**11: 6** (5, 1)	287
Integard Pro	1: 1	5,137	4,027 (78.4: 70.5 7.9)	**16: 10** (9, 1)	322,559
Mplayer Lite	5: 2	117,822	104,671 (88.8: 70.0 18.8)	**18: 7** (6, 1)	1,128,959
msvcr71.dll	1: 1	10,301	7,129 (69.2: 59.6 9.6)	**14: 9** (8, 1)	3,317,760
msvcr71.dll	1: 1	10,301	7,129 (69.2: 59.6 9.6)	**16: 8** (8, 0)	1,728,000
mscorie.dll	1: 1	1,616	1,304 (80.6: 73.5 7.1)	**10: 4** (4, 0)	25,200
mfc71u.dll	1: 1	86,803	64,053 (73.8: 68.7 5.1)	**11: 6** (6, 0)	170,496

for the completion of all tests by taking the average time across multiple runs, using both the original and the randomized versions of the DLLs. In all cases, there was no observable difference in the two times, within measurement error.

3.3.3 Effectiveness

3.3.3.1 ROP Exploits

We evaluated the effectiveness of in-place code randomization using publicly available ROP exploits against vulnerable Windows applications [Metasploit, 2010; Node, 2010; Nate_M, 2011], as well as generic ROP payloads based on commonly used DLLs [Immunity, 2010; Corelan Team, a]. These seven different ROP code implementations, listed in Table 3.2, bypass Windows DEP and execute a second-stage shellcode, as described in Sec. 2.1, and work even in the latest version of Windows, with DEP and ASLR enabled. The ROP code used in the three exploits is implemented with gadgets from one or a few DLLs that do not support ASLR, as shown in the second column of Table 3.2. The number of unique gadgets used in each case varies between 10–18, and typically a large part of the gadgets is repeatedly executed at many points throughout the ROP code. When replacing the original non-ASLR DLLs of each application with randomized versions, in all cases the exploits were rendered unsuccessful. Similarly, we used a custom application to test the generic ROP payloads and verified that the ROP code did not succeed when the corresponding DLL was randomized.

The ROP code of the exploit against Acrobat Reader uses just 11 unique gadgets, all coming from a single non-ASLR DLL (icucnv36.dll). From these gadgets, in-place code randomization can alter six of them: one gadget is completely eliminated, while the other five broken gadgets have 2, 2, 3, 4, and 6 possible states, respectively, resulting to a total of 287 randomized states (*in addition* to the always eliminated gadget, which also alone breaks the ROP code). Even if we assume that no elimination were possible, the exploit would still succeed only in one out of the 288 (0.35%) possible instances (including the original) of the given gadget set. Considering that this is a client-side exploit, in which the attacker will probably have only one or a few opportunities for tricking the user to open the malicious PDF file, the achieved randomization entropy is quite high—always assuming that none of the gadgets could have been eliminated. As shown in Table 3.2, the number of possible randomized states in the rest of the cases is several orders of magnitude higher. This is mostly due to the larger number of broken gadgets, as well as due to a few broken gadgets with tens of possible modified states, which both increase the number of states exponentially.

Next, we explored whether the affected gadgets could be directly replaced with unmodifiable gadgets in order to reliably circumvent our technique. Out of the six affected gadgets in the Adobe Reader exploit, only four can be directly replaced, meaning that the exploit cannot be trivially modified to bypass randomization. Furthermore, two of the gadgets have only one replacement each, and both replacements are found in code regions that are not discovered by IDA Pro—both could be randomized using a more precise code extraction method. For the rest of the ROP payloads, there are at least three irreplaceable gadgets in each case.

We should note that the relatively small number of gadgets used in most of these ROP payloads is a worst-case scenario for our technique, which however not only is able to prevent these exploits, but also does not allow the attacker to directly replace all the affected gadgets. Indeed, besides the more complex ROP payloads used in the Integard and Mplayer exploits, the rest of the payloads use API functions that are already imported by a non-ASLR DLL, and simply call them directly using hard-coded addresses. This type of API invocation is much simpler and requires fewer gadgets [Schwartz *et al.*, 2011] compared to ROP code like the one used in the Integard and Mplayer exploits (16 and 18 unique gadgets, respectively), which first dynamically locates a pointer to kernel32.dll (always ASLR-enabled in Windows 7) and then gets a handle to `VirtualProtect`.

3.3.3.2 Automated ROP Payload Generation

The fact that some of the randomized gadgets are not directly replaceable does not necessarily mean that the same outcome cannot be achieved using solely unmodifiable gadgets. To assess whether an attacker could construct a ROP payload resistant to in-place code randomization based on gadgets that cannot be randomized, we used Q [Schwartz *et al.*, 2011] and Mona [Corelan Team, b], two automated ROP code construction tools.

Q is a general-purpose ROP compiler that uses semantic program verification techniques to identify the functionality of gadgets, and provides a custom language, named QooL, for writing input programs. Its current implementation only supports simple QooL programs that call a single function or system call, while passing a single custom argument. In case the function to be called belongs to an ASLR-enabled DLL, Q can compute a handle to it through the import table of a non-ASLR DLL [Fresi Roglia *et al.*, 2009], when applicable. We should note that although Q currently compiles only basic QooL programs that call a single API function, this does not limit our evaluation, but on the contrary, stresses even more our technique. The simpler the programs, the fewer the gadgets used, which makes it easier for Q to generate ROP code even when our technique limits the number of available gadgets.

Mona is a plug-in for Immunity Debugger [Immunity,] that automates the process of building Windows ROP payloads for bypassing DEP. Given a set of non-ASLR DLLs, Mona searches for available gadgets, categorizes them according to their functionality, and then attempts to automatically generate four alternative ROP payloads for giving execute permission to the embedded shellcode and then invoking it, based on the `VirtualProtect`, `VirtualAlloc`, `NtSetInformationProcess`, and `SetProcessDEPPolicy` API functions (the latter two are not supported in Windows 7).

Considering the functionality of the ROP payloads generated by the two tools, Mona generates slightly more complex payloads, but its gadget composition engine is less so-

Table 3.3: Results of running Qand Monaon the original non-ASLR DLLs listed in Table 3.2, and the unmodified parts of their randomized versions. In all cases, both tools failed to generate a ROP payload using solely non-randomized gadgets.

	Q success		Mona success	
Application/DLL	Orig.	Rand.	Orig.	Rand.
Adobe Reader	✔	✘	✔ (VA)	✘
Integard Pro	✔	✘	✘	✘
Mplayer	✔	✘	✔ (VA)	✘
msvcr71.dll	✔	✘	✘	✘
mscorie.dll	✘	✘	✘	✘
mfc71u.dll	✔	✘	✔ (VA,VP)	✘

phisticated compared to Q's. Q generates payloads that compute a function address, construct its single argument, and call it. Payloads generated by Mona also call a single memory allocation API function (which though requires the construction of several arguments), copy the shellcode to the newly allocated area, and transfer control to it. Note that the complexity of the ROP code used in the tested exploits is even higher, since they rely on up to four different API functions [Metasploit, 2010], or "walk up" the stack to discover pointers to non-imported functions from ASLR-enabled DLLs [Node, 2010; Nate_M, 2011].

Table 3.3 shows the results of running Q and Mona on the same set of applications and DLLs used in the previous section (for applications, all non-ASLR DLLs are analyzed collectively), for two different cases: when all gadgets are available to the ROP compiler, and when only the non-randomized gadgets are available. The second case aims to build a payload that will be functional even when code randomization is applied. Although both Q and Mona were able to create payloads when applied on the original DLLs in almost all cases, they failed to construct any payload using only non-randomized gadgets in *all* cases.

Although our technique was able to prevent two different tools from automatically constructing reliable ROP code, this favorable outcome does not exclude the possibility that a functional payload could still be constructed based solely on non-randomized gadgets, e.g., in a manual way or using an even more sophisticated ROP compiler. However, it clearly

Table 3.4: Number of useful gadgets identified by Qin the original code segments / in their unmodifiable parts after in-place randomization was applied.

Gadget Type	Reader	Integard	Mplayer	msvcr71	mscorie	mfc71u	total	(%)
Pivots	171/27	55/11	156/48	89/18	13/5	65/20	549/129	(23.50)
Storemem	162/11	14/4	105/7	33/6	1/1	69/15	384/44	(11.46)
Move	57/7	25/13	68/35	31/12	7/3	62/60	250/130	(52.00)
ArithmeticStore	89/8	7/3	90/6	31/4	-	16/8	233/29	(12.45)
ArithmeticLoad	587/23	26/8	1194/40	147/24	-	290/104	2244/199	(8.87)
JumpConsts	1/1	1/1	1/1	1/1	1/1	1/1	6/6	(100.00)
SwitchStack	171/27	55/11	156/48	89/18	13/5	65/20	549/129	(23.50)
Loadmem	657/79	18/0	314/129	71/36	-	761/690	1821/934	(51.29)
LoadConst	424/36	121/20	621/138	155/23	14/3	175/67	1510/287	(19.01)
Arithmetic	409/49	59/10	517/66	167/41	8/2	347/190	1507/358	(23.76)

demonstrates that in-place code randomization significantly raises the bar for attackers, and makes the construction of reliable ROP code much harder, even in an automated way.

This is reflected on the reduction in the number of available (non-randomized) gadgets after code randomization. Both tools operate in two phases: gadget discovery and code compilation. During the first phase, they search for useful gadgets and categorize them according to their functionality. Tables 3.4, 3.5 and 3.6. show the number of useful gadgets as reported by Q and Mona, respectively, that are available before and after randomization. As shown by the percentage of the remaining gadgets (last column), many gadget types have very few available gadgets or are eliminated completely, which makes the construction of reliable ROP code much harder.

3.4 Discussion

In-place code randomization may not always randomize a significant part of the executable address space, and it is hard to give a definitive answer on whether the remaining unmodifiable gadgets would be sufficient for constructing useful ROP code. This depends on the code in the non-ASLR address space of the particular vulnerable process, as well as on the actual operations that need to be achieved using ROP code. Note that Turing-completeness is irrelevant for practical exploitation [Schwartz *et al.*, 2011], and none of the gadget sets used in the tested ROP payloads is Turing-complete. For this reason, we emphasize that in-place code randomization should be used as a mitigation technique, in the same fashion as application armoring tools like EMET [Microsoft, a], and not as a complete prevention solution.

As previous studies [Schwartz *et al.*, 2011; Shacham, 2007; Dullien *et al.*, 2010] have shown, though, the feasibility of building a ROP payload is proportional to the size of the non-ASLR code base, and reversely proportional to the complexity of the desired functionality. Our experimental evaluation shows that in all cases, the space of the remaining useful gadgets after randomization is sufficiently small to prevent the automated generation of a ROP payload. At the same time, the tested ROP payloads are far from the complexity of a fully blown ROP-based implementation of the operations required for carrying out an attack, such as dumping a malicious executable on disk and executing it. Currently, this functionality is handled by the embedded shellcode, which in essence allows us to view these ROP payloads as sophisticated versions of return-to-libc. We should stress that the randomization coverage of our prototype implementation is a lower bound for what would be possible using a more sophisticated code extraction method [Nanda *et al.*, 2006; Smithson *et al.*, 2010]. In our future work, we also plan to relax some of the conservative assumptions that we have made in instruction reordering and register reassignment, using data flow analysis based on constant propagation.

Given its practically zero overhead and direct applicability on third-party executables, in-place code randomization can be readily combined with existing techniques to improve diversity and reduce overheads. For instance, compiler-level techniques against ROP attacks [Li *et al.*, 2010; Onarlioglu *et al.*, 2010] increase significantly the size of the generated code, and also affect the runtime overhead. Incorporating code randomization for eliminating some of the gadgets could offer savings in code expansion and runtime overheads. Our technique is also applicable in conjunction with randomization methods based on code block reordering [Forrest *et al.*, 1997; Bhatkar *et al.*, 2005; Kil *et al.*, 2006], to further increase randomization entropy.

In-place code randomization at the binary level is not applicable for software that performs self-checksumming or other runtime code integrity checks. Although not encountered in the tested applications, some third-party programs may use such checks for hindering reverse engineering. Similarly, packed executables cannot be modified directly. However, in most third-party applications, only the setup executable used for software distribution is packed, and after installation all extracted PE files are available for randomization.

Code randomization in general is not designed to prevent against attackers that have access to the randomized code. As it was recently shown, such attacks are practical in some environments (e.g., in the browser) [Snow *et al.*, 2013]. It is trivial to see that this kind of attacks could be prevented by setting the permissions of code regions to execute-only (and not readable). This, however, is not easy to accomplish in some architectures like x86 where the execute permission implies readability. Still, making code regions unreadable is possible in recent x86 chips by using the Extended Page Tables (EPT) [Intel, 2014, Sec. 28.2] or by using a technique called Split TLB [Wurster *et al.*, 2005]. A more generic approach, with added benefits, would be to combine in-place code randomization with Instruction Set Randomization (ISR) [Kc *et al.*, 2003; Barrantes *et al.*, 2003].

Finally, although quite effective as a standalone mitigation, in-place code randomization is not meant to be a complete prevention solution, as it offers probabilistic protection and thus cannot deliver any protection guarantees. However, it can be applied in tandem with existing randomization techniques to increase process diversification. This is facilitated by the practically zero overhead of the applied transformations, and the ease with which they can be applied on existing third-party executables. Section 4.6 describes the benefits of combining it with our second technique.

Table 3.5: Number of useful gadgets identified by Monain the original code segments / in their unmodifiable parts after in-place randomization was applied.

Gadget Type	Reader	Integard	Mplayer	msvcr71	mscorie	mfc71u	total	(%)
add eax -> ebx	–	–	–	3/0	–	–	3/0	(0.00)
add ebp -> eax	1/0	–	–	–	–	1/0	2/0	(0.00)
add ebp -> ebx	–	–	–	–	–	1/1	1/1	(100.00)
add ebp -> edi	–	–	–	–	–	1/1	1/1	(100.00)
add ebp -> edx	–	–	2/0	–	–	–	2/0	(0.00)
add ebx -> eax	1/0	–	4/0	–	–	–	5/0	(0.00)
add ebx -> ecx	–	–	1/0	–	–	–	1/0	(0.00)
add ebx -> edx	1/0	1/1	1/0	1/0	1/1	1/0	6/2	(33.33)
add ecx -> eax	5/0	–	4/0	2/0	–	–	11/0	(0.00)
add ecx -> ebp	–	–	3/1	–	–	–	3/1	(33.33)
add edi -> eax	4/0	–	3/0	–	–	1/0	8/0	(0.00)
add edi -> ecx	–	–	8/0	–	–	–	8/0	(0.00)
add edi -> edx	–	–	4/0	–	–	–	4/0	(0.00)
add edx -> eax	3/0	–	5/0	–	–	–	8/0	(0.00)
add esi -> eax	9/0	–	5/0	2/0	–	–	16/0	(0.00)
add esi -> ecx	–	–	16/0	–	–	–	16/0	(0.00)
add esi -> edi	–	–	3/0	–	–	4/4	7/4	(57.14)
add value to eax	3/2	2/1	2/2	2/2	2/1	4/1	15/9	(60.00)
add value to ebx	1/0	–	–	–	–	–	1/0	(0.00)
add value to edi	–	–	–	–	–	1/0	1/0	(0.00)
add value to edx	–	–	1/0	–	–	–	1/0	(0.00)
add value to esi	–	–	–	–	–	1/0	1/0	(0.00)
dec eax	24/9	–	84/24	22/4	1/1	33/11	164/49	(29.88)
dec ebp	–	–	3/0	–	–	1/1	4/1	(25.00)
dec ebx	–	–	4/3	–	–	–	4/3	(75.00)
dec ecx	2/0	5/5	18/12	63/59	1/1	186/177	275/254	(92.36)
dec edi	2/0	–	2/0	–	–	2/2	6/2	(33.33)
dec edx	111/87	–	3/2	1/1	–	3/2	118/92	(77.97)
dec esi	1/0	–	5/3	1/0	–	1/0	8/3	(37.50)
empty eax	156/2	1/0	133/0	89/0	5/0	196/4	580/6	(1.03)
empty edi	–	–	–	1/0	–	–	1/0	(0.00)
empty edx	2/0	–	2/0	–	–	5/0	9/0	(0.00)
inc eax	51/19	2/0	53/6	108/95	9/2	281/141	504/263	(52.18)
inc ebp	–	–	134/0	1/1	–	2/1	137/2	(1.46)
inc ebx	6/2	–	9/3	12/1	–	6/1	33/7	(21.21)
inc ecx	0/5	–	9/0	1/0	–	12/8	22/13	(59.09)
inc edi	3/0	–	1/1	2/0	–	7/0	13/1	(7.69)
inc edx	3/0	–	37/1	1/0	–	1/1	42/2	(4.76)
inc esi	14/1	1/0	2/0	3/0	–	11/1	31/2	(6.45)
move eax -> ebp	3/0	–	23/2	–	–	5/1	31/3	(9.68)
move eax -> ebx	–	–	52/0	3/0	–	2/1	57/1	(1.75)
move eax -> ecx	1/0	1/1	7/1	–	–	–	9/2	(22.22)
move eax -> edi	4/0	–	7/0	–	–	17/1	28/1	(3.57)
move eax -> edx	–	–	10/1	–	–	–	10/1	(10.00)
move eax -> esi	2/0	–	19/0	–	–	13/1	34/1	(2.94)
move eax -> esp	11/2	–	30/3	1/0	–	43/23	85/28	(32.94)
move ebp -> eax	34/0	–	80/2	2/0	–	17/1	133/3	(2.26)
move ebp -> ebx	–	–	2/0	–	–	1/1	3/1	(33.33)

Table 3.6: Continuation of Talbe 3.5

Gadget Type	Reader	Integard	Mplayer	msvcr71	mscorie	mfc71u	total	(%)
move ebp -> edi	5/0	–	2/0	–	–	2/1	9/1	(11.11)
move ebp -> edx	–	–	6/0	–	–	–	6/0	(0.00)
move ebx -> eax	96/0	2/0	151/0	8/0	1/0	37/1	295/1	(0.34)
move ebx -> ecx	–	–	1/0	–	–	–	1/0	(0.00)
move ebx -> edi	1/0	–	–	–	–	3/0	4/0	(0.00)
move ebx -> edx	1/0	1/1	1/0	1/0	1/1	1/0	6/2	(33.33)
move ebx -> esp	4/0	–	2/0	2/1	–	–	8/1	(12.50)
move ecx -> eax	26/1	3/2	46/1	10/4	1/0	41/2	127/10	(7.87)
move ecx -> ebp	–	–	3/1	–	1/0	3/1	7/2	(28.57)
move ecx -> ebx	–	–	4/0	–	–	–	4/0	(0.00)
move ecx -> edi	–	–	1/0	–	–	6/0	7/0	(0.00)
move ecx -> edx	2/0	–	–	–	–	2/0	4/0	(0.00)
move ecx -> esi	1/0	–	–	–	–	5/0	6/0	(0.00)
move ecx -> esp	–	–	–	–	–	2/0	2/0	(0.00)
move edi -> eax	125/0	–	92/8	15/0	6/0	96/1	334/9	(2.69)
move edi -> ebp	1/0	–	–	–	–	–	1/0	(0.00)
move edi -> ebx	–	–	1/0	–	–	–	1/0	(0.00)
move edi -> ecx	1/0	–	8/0	–	–	–	9/0	(0.00)
move edi -> edx	–	–	19/0	–	–	–	19/0	(0.00)
move edi -> esi	–	–	3/0	–	–	5/5	8/5	(62.50)
move edi -> esp	–	–	19/0	–	–	–	19/0	(0.00)
move edx -> eax	17/1	–	92/1	1/0	1/1	6/0	117/3	(2.56)
move edx -> ebx	1/0	–	3/0	–	–	–	4/0	(0.00)
move edx -> ecx	–	–	–	1/0	–	–	1/0	(0.00)
move edx -> edi	1/0	–	–	–	–	1/0	2/0	(0.00)
move edx -> esi	1/0	–	–	–	–	1/0	2/0	(0.00)
move esi -> eax	488/0	2/0	136/0	58/0	12/1	513/2	1209/3	(0.25)
move esi -> ebx	–	–	2/0	–	–	–	2/0	(0.00)
move esi -> ecx	2/0	–	16/0	–	–	–	18/0	(0.00)
move esi -> edi	–	–	3/0	–	–	4/4	7/4	(57.14)
move esi -> edx	–	–	8/0	–	–	–	8/0	(0.00)
move esi -> esp	1/0	–	17/0	–	–	–	18/0	(0.00)
move esp -> eax	1/0	–	1/0	–	–	–	2/0	(0.00)
move esp -> ebp	–	–	1/0	–	–	–	1/0	(0.00)
move esp -> ebx	5/0	–	85/0	–	–	–	90/0	(0.00)
move esp -> ecx	8/0	–	–	1/0	–	–	9/0	(0.00)
move esp -> edi	37/0	–	10/0	–	–	2/0	49/0	(0.00)
move esp -> esi	20/0	–	4/0	2/0	–	5/0	31/0	(0.00)
neg eax	3/1	–	1/1	7/0	–	9/8	20/10	(50.00)
neg edx	–	–	–	1/0	–	–	1/0	(0.00)
put ptr into eax	15/10	1/0	12/3	2/1	–	23/15	53/29	(54.72)
put ptr into ecx	–	–	2/0	–	–	–	2/0	(0.00)
pushad	7/0	–	26/4	1/0	–	17/12	51/16	(31.37)
xor ebp -> eax	1/0	–	–	–	–	–	1/0	(0.00)
xor edx -> eax	–	–	1/0	–	–	–	1/0	(0.00)
xor esi -> eax	1/0	–	–	–	–	–	1/0	(0.00)

Chapter 4

Indirect Branch Tracing

As previously stated in Section 1.2, transparency is a key factor for enabling the practical applicability of techniques that aim to protect proprietary software. The absence of any need for modifications to existing binaries ensures an easy deployment process, and can even enable the protection of applications that are already installed on end-user systems [Microsoft, a]. At the same time, to be practical, mitigation techniques should introduce minimal overhead, and should not affect the proper execution of the protected applications due to incompatibility issues or false positives.

Aiming to fulfill the above requirements, in this chapter we present a fully transparent runtime ROP exploit mitigation technique for the protection of third-party applications. Our approach is based on monitoring the executed indirect branches at critical points during the lifetime of a process, and identifying abnormal control flow transfers that are inherently exhibited during the execution of ROP code. The technique is built around Last Branch Recording (LBR), a recent feature of Intel processors. Relying mainly on hardware for instruction-level monitoring allows for minimal runtime overhead and completely transparent operation, without requiring any modifications to the protected applications.

Our current design performs two checks on indirect branches. First, it restricts the targets of return instructions to leginimate code locations. Second, it checks for sequences of relatively short code fragments chained through any kind of indirect branches. The first check detects traditional ROP attacks (i.e., using gadgets ending in return instructions), whereas the second is more general and detects even ROP code that uses gadgets ending with indirect `jmp` or `call` instructions. We have developed a prototype implementation for Windows 7, named *kBouncer*. To minimize context switching overhead, branch analysis is performed only before critical system operations that could cause any harm. We evaluated the effectiveness and practical applicability of our technique using publicly available ROP exploits against widely used software, including Internet Explorer, Adobe Flash Player, and Adobe Reader. In all cases, kBouncer blocks the exploit successfully, and notifies the user through a standard error message window. We also verified that kBouncer introduces minimal overhead by stress-testing our implementation with workloads that trigger excessively the protected system.

4.1 Practical Indirect Branch Tracing for ROP Prevention

The proposed approach uses runtime process monitoring to block the execution of code that exhibits return-oriented behavior. In contrast to typical program code, the code used in ROP exploits consists of several small instruction sequences, called *gadgets*, scattered through the executable segments of the vulnerable process. Gadgets end with an indirect branch instruction that transfers control to the following gadget according to a sequence of gadget addresses contained in the "payload" that is injected during the attack. As the name of the technique implies, gadgets typically end with a `ret` instruction, although any combination of indirect control transfer instructions can be used [Checkoway *et al.*, 2010].

The key observation behind our approach is that the execution behavior of ROP code has some inherent attributes that differentiate it from the execution of legitimate code. By monitoring the execution of a process while focusing on those properties, kBouncer can identify and block a ROP exploit before its code accomplishes any critical operation.

In this section, we discuss in detail how kBouncer leverages the Last Branch Recording feature of recent processors to retrieve the sequence of the most recent indirect branch instructions that took place right before the invocation of a system function. In the following section, we discuss how kBouncer uses this information to identify the execution of ROP code. As the vast majority of in-the-wild ROP exploits target Windows software, our design focuses on achieving transparent operation for existing Windows applications without

Table 4.1: Qualitative comparison of alternative techniques for runtime branch monitoring.

Technique	Overhead	Requirements	Compatibility	Deployment
Compiler-level	med	source	some	hard
Binary rewriting	med	pdb	no	med
Dynamic instrumentation	high	-	yes	med
LBR monitoring	low	-	yes	easy

raising any compatibility issues or false alerts.

4.1.1 Branch Tracing vs. Other Approaches

Execution monitoring at the instruction level usually comes with an increased runtime overhead. Even when tracking only a particular subset of instructions, e.g., in our case only indirect control transfer instructions, the overhead of interrupting the normal flow of control and updating the necessary accounting information is prohibitive for production systems. There are several different approaches that can be followed for monitoring the execution of indirect branch instructions, each of them having different requirements, performance overhead, transparency level, and deployment effort.

Extending the compiler to generate and embed runtime checks in the executable binary at compile time is one of the simplest techniques [Onarlioglu *et al.*, 2010]. However, the high frequency of control transfer instructions in typical code means that a lot of additional instrumentation code must be added. Also, deployment requires a huge effort as all programs have to be recompiled. Another option is static binary rewriting. Its main advantage over compiler-level techniques is that no source code is required, but only debug symbols (e.g., PDB files) [Abadi *et al.*, 2005]. Still, all control transfers need to be checked. Even worse, it breaks self-checksumming or signed code and cannot be applied to self-modifying programs. Dynamic binary instrumentation is another alternative that can handle even stripped binaries (no need for source code or debug symbols), but the runtime performance overhead of existing binary instrumentation frameworks slows down the normal execution of an application by a factor of a few times [Davi *et al.*, 2011].

In contrast to the above approaches, our system monitors the executed indirect branch instructions using Last Branch Recording (LBR) [Intel, 2014, Sec. 17.4], a recent feature of Intel processors introduced in the Nehalem architecture. When LBR is enabled, the CPU tracks the last N (16 for the CPU model we used) most recent branches in a set of 64-bit model-specific registers (MSR). Each branch record consists of two MSR registers, which hold the linear addresses of the branch instruction and its target instruction, respectively. Records from the LBR stack can be retrieved using a special instruction (`rdmsr`) from privileged mode. The processor can be configured to track only a subset of branches based on their type: relative/indirect calls/jumps, returns, and so on.

Table 4.1 shows a summarized comparison of the alternative strategies discussed above. For our particular case, the use of LBR has several advantages: it incurs zero overhead for storing the branches; it is fully transparent to the running applications; is does not cause any incompatibility issues as it is completely decoupled from the actual execution; it does not require source code or debug symbols; and it can be dynamically enabled for already installed applications—there is no need for recompilation or instruction-level instrumentation.

4.1.2 Using Last Branch Recording for ROP Prevention

Although the CPU continuously records the most recent branches in the LBR stack with zero overhead, accessing the LBR registers and retrieving the recorded information unavoidably adds some overhead. Considering the limited size (16 entries) of the LBR stack, and that it can be accessed only from kernel-level code, checking the targets of all indirect control transfer instructions would incur a prohibitively high performance overhead. Indirect branches occur very frequently in typical programs, and a monitored process should be interrupted once every 16 branches with a context switch. In fact, the implementation of such a scheme is not facilitated by the current design of the LBR feature, as it does not provide any means of interrupting execution whenever the stack gets full after retrieving its previous 16 records.

Fortunately, when considering the actual operations of a ROP exploit, it is possible to dramatically reduce the number of control transfer instructions that need to be inspected. The typical end goal of malicious code is to give the attacker full control of the victim

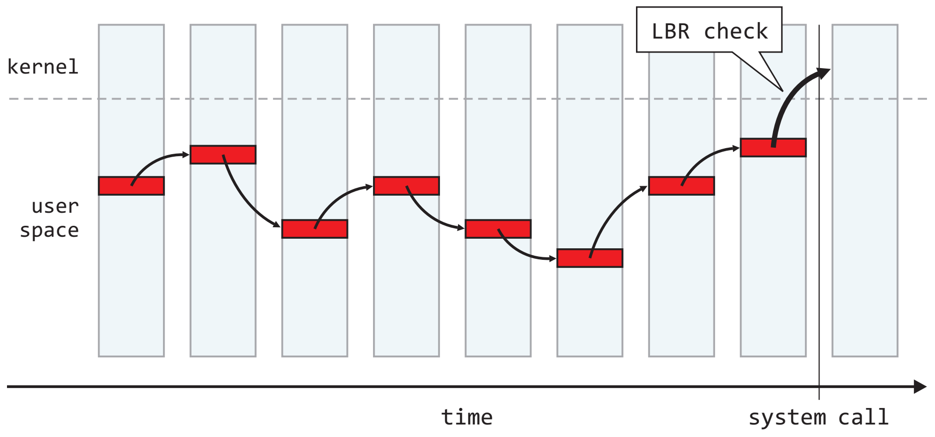


Figure 4.1: Illustration of a basic scheme for ROP code detection. Whenever control is transferred from user to kernel space (vertical line), the system inspects the most recent indirect branches to decide whether the system call was invoked by ROP code or by the actual program.

system. This usually involves just a few simple operations, such as dropping and executing a malicious executable on the victim system, which unavoidably require interaction with the OS through the system call interface. Based on this observation, we can refine the set of indirect branches that need to be inspected to only those along the final part of the execution path that lead to a system call invocation. (Depending on the vulnerable program, exploitation might be possible without invoking any system call, e.g., by modifying a user authentication variable [Chen *et al.*, 2005], but such attacks are rarely found in the client-side applications that are typically targeted by current ROP exploits, and are outside the scope of this work.)

Figure 4.1 illustrates this approach. Vertical bars correspond to snapshots of the address space of a process, and arrows correspond to indirect control transfers. The vertical line denotes the point at which the flow of control is transferred from user space to kernel space through a system call. At this point, by interposing at the OS's system call handler, the system can access the LBR stack and retrieve the targets of the indirect branches that led to the system call. It can then check the control flow path for abnormal control transfers and distinctive properties of ROP-like behavior using the techniques that will be described in Sec. 4.2, and decide whether the system call is part of malicious ROP code, or it is being invoked legitimately by the actual program.

4.1.2.1 System Calls vs. API Calls

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [Microsoft, d], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [Russinovich, 2006], implemented in `ntdll.dll`, to invoke kernel-level services.

Exploit code rarely relies on the Native API for several reasons. One problem is that system call numbers change between Windows versions and service pack levels [Bania, 2005; Jurczyk, 2011], reducing the reliability of the exploit across different targets (or increasing attack complexity by having to adjust the exploit according to the victim's OS version). Most importantly, the desired functionality is often not conveniently exposed at all through the Native API, as for example is the case with the socket API [Skape, 2003]. Typically, the purpose of ROP code is to give execute permission to a second-stage shellcode using `VirtualProtect` or a similar API function [Erlingsson, 2007; Corelan Team, b; Metasploit, 2010; Nate_M, 2011; Metasploit, 2012b; Metasploit, 2012a]. The second-stage shellcode can be avoided altogether by implementing all the necessary functionality solely using ROP code, as is the case with a recent exploit against Adobe Reader XI, in which the ROP code calls directly the `fsopen`, `write`, `fclose`, and `LoadLibraryW` functions to drop and execute a malicious DLL [Bennett *et al.*, 2013].

The implementation of many of the functions exported by the Windows API is quite complex, and often involves several internal functions that are executed before the invocation of the intended system call. Due to the limited size of the LBR stack, this means that by the time execution reaches the actual system call, the LBR stack might be filled with indirect branches that took place *after* the Windows API function was called. To assess the extent of this effect, we measured the average number of indirect branch instructions (`ret`,

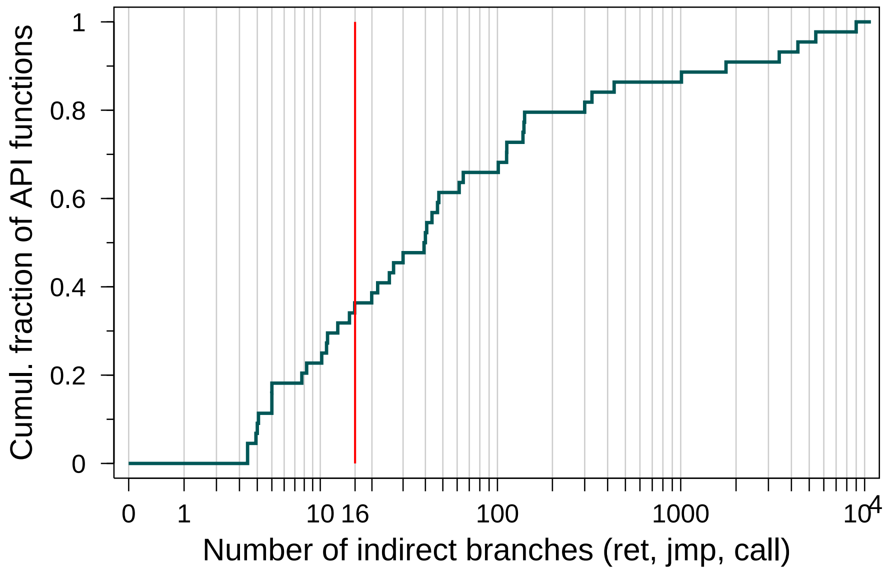


Figure 4.2: LBR overwriting due to indirect branches that take place within Windows API functions, prior to the execution of a system call.

`jmp`, and `call`) that are executed between the first instruction of a Windows API function and the system call it invokes, for a set of 52 "sensitive" functions that are commonly used in Windows shellcode and ROP code implementations (a complete list of the tested functions is provided in the appendix). As shown in Fig. 4.2, about 34% of the API functions execute less that 16 indirect branches, while the rest of them completely overwrite the LBR stack.

As these branches are made as part of legitimate execution paths, calling a function that completely overwrites the LBR stack would allow ROP code to evade detection. However, this scheme can be improved to provide robust detection of ROP code that calls *any* sensitive API function, irrespectively of the extent of overwriting in the LBR stack due to code in the function body.

4.1.2.2 LBR Stack Inspection on API Function Entry

Given that i) exploit code usually calls Windows API functions instead of directly invoking system calls, and ii) most API functions overwrite the LBR stack with legitimate indirect branches before invoking a system call, kBouncer inspects the LBR stack at the time an

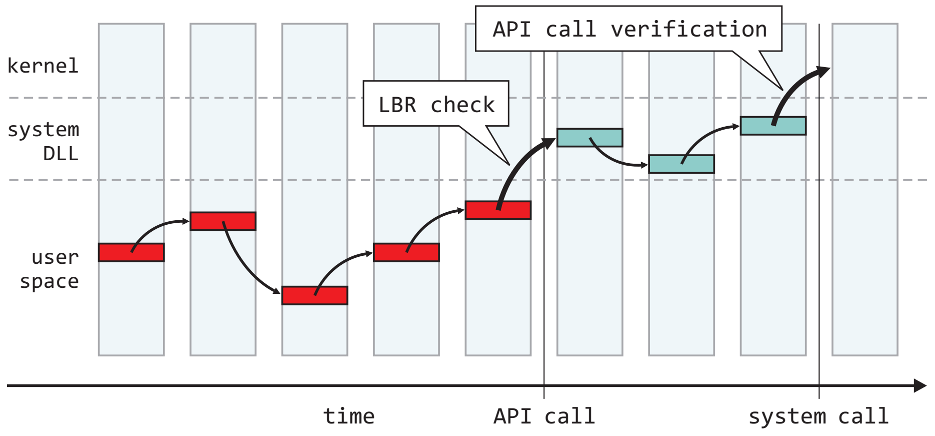


Figure 4.3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer's function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

API function is called, instead upon system call invocation. This allows the detection of ROP code that uses any sensitive API function, irrespectively of the number of legitimate indirect branches executed within its body. In case an API function is called by ROP code, all entries in the LBR stack at the time of function entry will correspond to the indirect branches of the gadgets that lead to the function call, as depicted in Fig. 4.3.

Still, without any additional precautions, this scheme would allow an attacker to bypass the LBR check at the entry point of a function. An implementation of the LBR check in the system call handler—within the kernel—safeguards it from user-level code and any bypass attempt. In contrast, implementing the LBR check as a hook to a user-level function's entry point does not provide the same level of protection. An attacker could avoid the check by jumping over the hook at the function's prologue, instead of jumping at its main entry point, and then normally executing the function body. Alternatively, by trading off some of its reliability, the ROP code could avoid calling the API function altogether by invoking directly the relevant Native API call.

Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are *always* invoked solely through their respective Windows API functions. After a clear LBR check at an API function's entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function, and clears it. If the checkpoint was not set, then this means that the flow of control did not pass through the proper API function preamble, and kBouncer reports a violation.

We should note that user-level ROP code cannot bypass kBouncer's checks by faking a checkpoint. The code for setting a checkpoint can only run with kernel privileges, and the checkpoint itself is stored in kernel space so that i) the system call handler can later access it, and ii) any user-level code (and consequently the ROP code itself) cannot tamper with it. The checkpoint code is tied with and comes right after the code that inspects the LBR stack, and both run in an atomic way at kernel level, i.e., the checkpoint cannot be set without previously analyzing the LBR for the presence of ROP code. This prevents any ROP code from faking a checkpoint without being detected—the part of the ROP code with the task of setting the checkpoint would be detected by the LBR check before the checkpoint is actually set.

4.2 Identifying the Execution Behavior of ROP Code

Before allowing a Windows API function call to proceed, kBouncer analyzes the most recent indirect branches that were recorded in the LBR cache prior to the function call. LBR is configured to record only `ret`, indirect `jmp`, and indirect `call` instructions. The execution of ROP code is identified by looking for two prominent attributes of its runtime behavior: i) illegal `ret` instructions that target locations not preceded by call sites, and ii) sequences of relatively short code fragments "chained" through any kind of indirect branches.

Returns that do not transfer control right after call sites is an illegitimate behavior exhibited by all publicly available ROP exploits against Windows software, which rely mainly on gadgets ending with `ret` instructions (`ret` conveniently manipulates both the

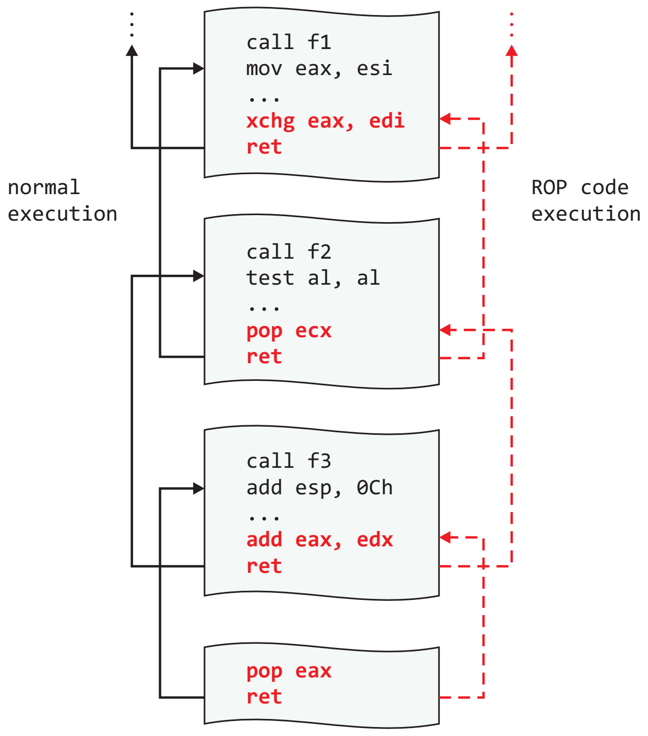


Figure 4.4: In normal code, `ret` instructions target valid call sites (left), while in ROP code, they target gadgets found in arbitrary locations (right).

program counter and the stack pointer). The second, more generic attribute captures an inherent property of not only purely return-oriented code, but also of advanced (and admittedly harder to construct) jump-oriented code (or even "hybrid" ROP/JOP code that might use any combination of gadgets ending with `jmp`, `call`, and `ret` instructions).

4.2.1 Illegal Returns

When focusing on the control flow behavior of ROP code at the instruction level, we expect to observe the successive execution of several `ret` instructions, which correspond to the transfer of control from each gadget to the next one. Although this control flow pattern is quite distinctive, the same pattern can also be observed in legitimate code, e.g., when a series of functions consecutively return to their callers. However, when considering the *targets* of `ret` instructions, there is a crucial difference.

In a typical program, `ret` instructions are paired with `call` instructions, and thus the target of a legitimate `ret` corresponds to the location right after the call site of the respective caller function, i.e., an instruction that follows a `call` instruction, as illustrated in the left part of Fig. 4.4. In contrast, a `ret` instruction at the end of a gadget transfers control to the first instruction of the following gadget, which is unlikely to be preceded by a `call` instruction. This is because gadgets are found in arbitrary locations across the code image of a process, and often may correspond to non-intended instruction sequences that happen to exist due to overlapping instructions [Shacham, 2007].

At runtime, the `ret` instructions of ROP code can be easily distinguished from the legitimate return instructions of a benign program by checking their targets. A `ret` instruction that transfers control to an instruction not preceded by a `call` is considered illegal, and the observation of an illegal `ret` is flagged by kBouncer as an indication of ROP code execution.

Ensuring `call`-`ret` pairing by verifying caller-callee semantics, e.g., using a shadow stack [Davi *et al.*, 2011], constrains the control flow of a process in a much stricter way than the proposed scheme. In practice, though, enforcing such a strict policy is problematic, due to the use of `setjmp`/`longjmp` constructs, `call`/`pop` "getPC" code commonly found in position-independent executables, tail call optimizations, and lightweight user-level threads such as Windows fibers, in which the context switch function called by the current thread returns to the thread that is scheduled next.

Instead of enforcing a strict control flow, kBouncer simply makes sure that `ret` instructions always target *any* among all valid call sites (even those that correspond to non-intended `call` instructions). This is a more relaxed constraint that is not expected to be violated (and which did not, for the set of applications tested as part of our experimental evaluation) even in programs that use constructs like the above. Its implementation is also much simpler, as there is no need to track the execution of `call` instructions—checking that the target of each `ret` falls right after a `call` is enough.

Call-preceded Gadgets Although the above scheme prohibits the execution of illegal returns, which are prominently exhibited by typical ROP exploits, an attacker might still be able to construct functional ROP code using gadgets that begin right after `call` instructions, to which we refer as *`call`-preceded gadgets*. Note that `call`-preceded gadgets may

Table 4.2: Details about the dataset used for gadget analysis.

		Indirect Branches			System	Protected
Application	Workload	jump	call	ret	Calls	API Calls
Windows Media Player	Music playback for ~30 secs	7.3M	7.5M	30.0M	196K	5150
Internet Explorer 9	Browse to google.com	3.4M	4.8M	17.7M	87K	7092
Adobe Flash Player	Watch a youtube video	9.6M	21.7M	94.1M	317K	46658
Microsoft Word	Scroll through a document	3.3M	11.5M	38.8M	178K	5425
Microsoft Excel	Open a rich spreadsheet	6.3M	18.1M	54.5M	212K	3957
Microsoft PowerPoint	View a presentation	10.2M	19.3M	68.8M	275K	6577
Adobe Reader XI	Scroll through a few pages	9.7M	19.5M	100.6M	101K	5026

begin after either intended or unintended `call` instructions. As kBouncer cannot know which `call` instructions were actually emitted by the compiler, if any of the possible valid instructions immediately preceding the instruction at a target address is a `call` instruction, then that address may correspond to the beginning of a `call`-preceded gadget.

The observation of a `ret` that targets an instruction located right after a `call` is considered by kBouncer as normal, and thus ROP code comprising only `call`-preceded gadgets would not be identified based on the first ROP code attribute kBouncer looks for during branch analysis. Although such code would still be identified due to its "chained gadgets" behavior, which we will discuss below, we first briefly explore the feasibility of such an attempt.

For our analysis we use a set of typical Windows applications, detailed in Table 4.2. The data is collected using a purpose-built execution analysis framework, described in Sec. 4.3.2. We consider as a gadget any (intended or unintended) instruction sequence that ends with an indirect branch, and which does not contain any privileged or invalid instruction. In contrast to the gadget constraints typically considered in relevant studies [Shacham, 2007; Checkoway *et al.*, 2010; Schwartz *et al.*, 2011; Chen *et al.*, 2009; Yuan *et al.*, 2011; Pappas *et al.*, 2012; Hiser *et al.*, 2012; Wartell *et al.*, 2012] and the actual gadgets used in real exploits [Corelan Team, b; Bennett *et al.*, 2013; Metasploit, 2010; Nate_M, 2011; Metasploit, 2012b; Metasploit, 2012a], i.e., contiguous instruction sequences no longer than five instructions, we follow a more conservative approach and consider gadgets that i) may be *split* into several fragments due to internal conditional or unconditional relative jumps,

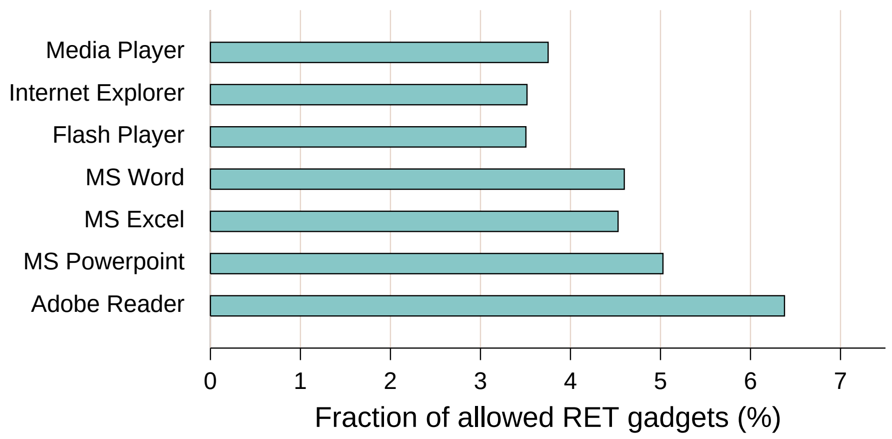


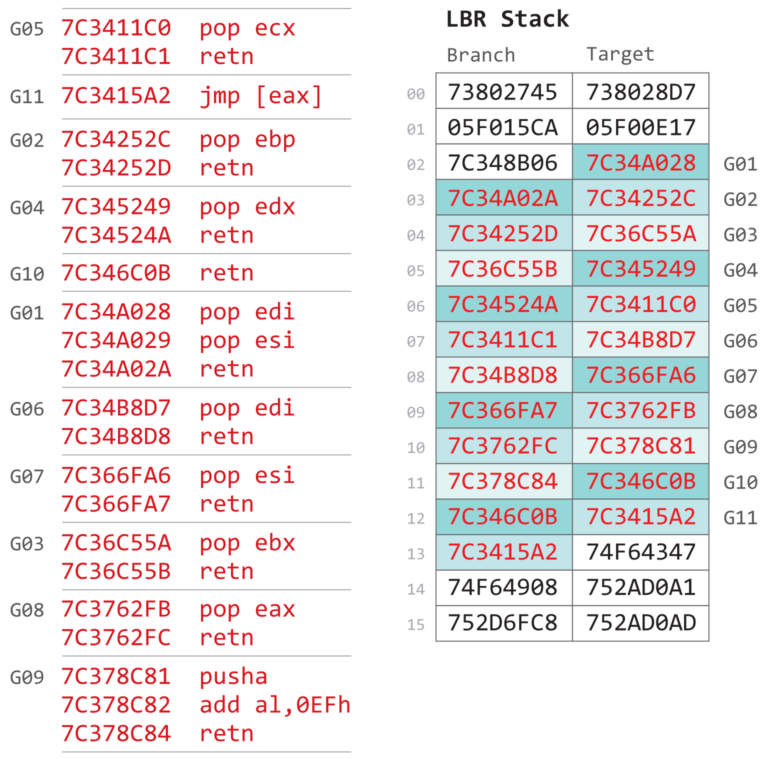
Figure 4.5: Among all gadgets that end with a `ret` instruction, only a small fraction (6.4% in the worst case for Adobe Reader) begin right after call sites.

and ii) have a maximum length of *20 instructions*.

Figure 4.5 shows the fraction of `call`-preceded gadgets among all gadgets that end with a `ret` instruction, for different Windows applications. In the worst case, only 6.4% of the gadgets begin right after call sites, a percentage much smaller compared to all available `ret` gadgets. Given that many of them are longer than the typical gadget size, and are thus harder to use in ROP code due to the many different operations and register or memory state changes they incur, an attacker would be left with a severely limited set of gadgets to work with. For comparison, the ROP payloads of the exploits we used in our evaluation, listed in Table 4.4, collectively use 44 unique gadgets with an average length of just 2.25 instructions, and only *three* of them happen to be `call`-preceded—the rest of them would all result in illegal returns.

4.2.2 Gadget Chaining

It is clear from the previous section that even a “lighter” version of kBouncer that would just prohibit the execution of illegal returns would still significantly raise the bar, as i) it would prevent the execution of the ROP code typically found in publicly available Windows exploits, and more importantly, ii) it would force attackers to either use only a limited set of `ret` gadgets, or resort to jump-oriented code—options of increased complexity.



```
G05 7C3411C0  pop ecx
    7C3411C1  retn
G11 7C3415A2  jmp [eax]
G02 7C34252C  pop ebp
    7C34252D  retn
G04 7C345249  pop edx
    7C34524A  retn
G10 7C346C0B  retn
G01 7C34A028  pop edi
    7C34A029  pop esi
    7C34A02A  retn
G06 7C34B8D7  pop edi
    7C34B8D8  retn
G07 7C366FA6  pop esi
    7C366FA7  retn
G03 7C36C55A  pop ebx
    7C36C55B  retn
G08 7C3762FB  pop eax
    7C3762FC  retn
G09 7C378C81  pusha
    7C378C82  add al,0EFh
    7C378C84  retn
```

LBR Stack

	Branch	Target	
00	73802745	738028D7	
01	05F015CA	05F00E17	
02	7C348B06	7C34A028	G01
03	7C34A02A	7C34252C	G02
04	7C34252D	7C36C55A	G03
05	7C36C55B	7C345249	G04
06	7C34524A	7C3411C0	G05
07	7C3411C1	7C34B8D7	G06
08	7C34B8D8	7C366FA6	G07
09	7C366FA7	7C3762FB	G08
10	7C3762FC	7C378C81	G09
11	7C378C84	7C346C0B	G10
12	7C346C0B	7C3415A2	G11
13	7C3415A2	74F64347	
14	74F64908	752AD0A1	
15	752D6FC8	752AD0AD	

Figure 4.6: The state of the LBR stack at the time kBouncer blocks an exploit against Adobe Flash.Diagonal pairs of addresses with the same shade correspond to the first and last instruction of each gadget.

To account for potential future exploits of these sorts, the second attribute that kBouncer uses to identify the execution of ROP code is an inherent characteristic of its construction: the observation of several short instruction sequences *chained* through indirect branches. This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP). Although legitimate programs also contain an abundance of code fragments linked with indirect branches, these fragments are typically much larger than gadgets, and more importantly, they do not tend to form long uninterrupted sequences (as we show below).

The CPU records in-sequence all executed indirect branches, enabling kBouncer to reconstruct the chain of gadgets used by any ROP code. Each LBR record $R[b, t]$ contains the address of the branch (b) and the address of its target (t), or from the viewpoint of ROP code, the *end* of a gadget and the *beginning* of the following one.

Figure 4.6 illustrates the contents of the LBR stack at the time kBouncer blocks the ROP code of an exploit against Adobe Flash [Metasploit, 2012a] (although kBouncer blocks this exploit due to illegal returns, we use it for illustrative purposes, as we are not aware of any publicly available JOP exploit). Starting with the most recent (bottom-most) record, the detection algorithm checks whether the target (located at address $R_{n-1}[t]$) of the previous branch, is an instruction that precedes the branch (located at address $R_n[b]$) of the current record. If starting from address $R_{n-1}[t]$, there exists an uninterrupted sequence of at most 20 instructions that ends with the indirect branch at address $R_n[b]$, then the sequence is considered as a gadget. Recall that kBouncer treats as gadgets even fragmented instruction sequences linked through conditional or unconditional relative jumps. The same process repeats with the previous records, moving upwards, as long as chained gadgets are found.

The ROP code in this example consists of 11 gadgets, all ending with a `ret` instruction except the final one (G11), which is a single-instruction gadget with an indirect `jmp` that transfers control to `VirtualProtect` in `kernel32.dll` (note the difference in the high bytes of the target address in record 13). The two bottom-most records in the LBR stack correspond to kBouncer's function hook (from `VirtualProtect` to `DeviceIoControl`, which signals the kernel component), and a `ret` from `__SEH_prolog4` which is called by `DeviceIoControl`.

A crucial question for the effectiveness of the above algorithm is whether legitimate code could be misclassified as ROP code due to excessively long chains of gadget-like instruction sequences. To assess this possibility, we measured the length of the gadget chains observed across all inspected LBR stack instances for the applications and workloads listed in Table 4.2. As described in Sec. 4.1.2.2, kBouncer inspects the LBR stack right before the execution of a sensitive Windows API function. In total, kBouncer inspected 79,885 LBR stack instances, i.e., the tested applications legitimately invoked a sensitive API function 79,885 times.

Figure 4.7 (solid line) shows the percentage of instances with a given gadget chain length. In the worst case, there is just one instance with a chain of five gadgets, and there are no instances with six or more gadgets. On the other hand, complex ROP code that would rely on `call`-preceded or non-`ret` gadgets would result in excessively long

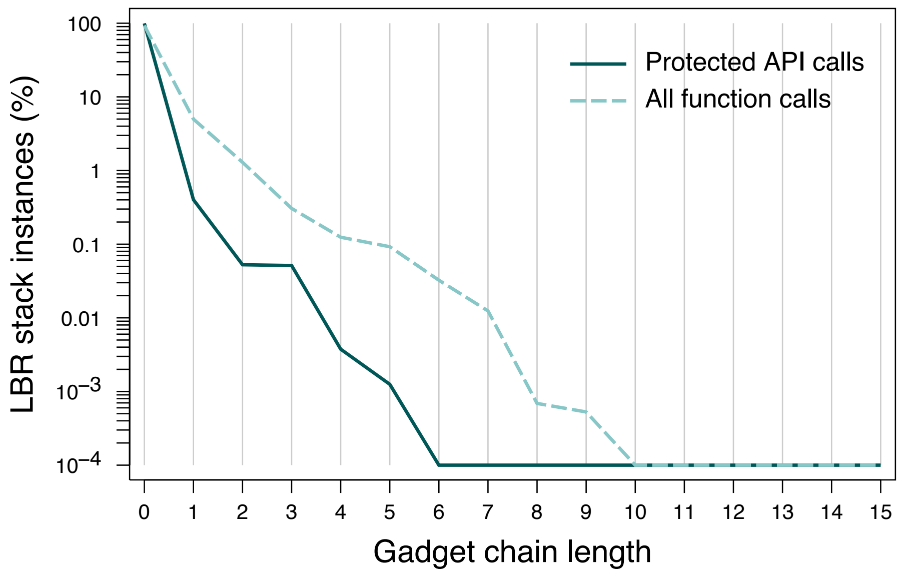


Figure 4.7: Percentage of LBR stack instances with a given gadget chain length for i) the instances inspected by kBouncer at the entry points of protected API function calls, and ii) the instances taken at the entry points of all function calls.

gadget chains, filling the LBR stack. Indicatively, a jump-oriented Turing-complete JOP implementation for Linux uses 34 gadgets [Checkoway *et al.*, 2010]. Furthermore, current JOP code implementations rely on a special dispatcher gadget that always executes between useful gadgets, at least doubling the amount of executed gadgets.

Although we can never rule out the possibility that benign code in some other application might result in a false positive, to ascertain that this possibility is unlikely, we also analyzed 97,554,189 LBR stack instances taken at the entry points of all executed functions during the lifetime of the same tested applications. In this orders-of-magnitude larger data set, the maximum gadget chain length observed is nine (dashed line), which is still far from filling up the LBR stack. This means that even if there is a need in the future to protect more API functions, or perform LBR checks in other parts of a program, we will more than likely still be able to set a robust detection threshold that will not result in false positives. For the current set of protected functions we use a threshold of eight gadgets, which allows for increased resilience to false positives.

Finally, note that in the above benign executions, the vast majority of the gadget-like chains stem from our conservative choice of considering *fragmented* gadgets of up to *20 instructions* long—significantly more complex and longer than the gadgets used in actual exploits. Although we could choose more reasonable constraints about what is considered as a gadget, we preferred to stress the limits of the proposed approach.

4.3 Implementation

4.3.1 kBouncer

To demonstrate the effectiveness of our proposed approach, we developed a prototype implementation for the x86 64-bit version of Windows 7 Professional SP1. Our prototype, kBouncer, consists of three components: i) an offline gadget extraction and analysis toolkit, ii) a user-space thin interposition layer between the applications and Windows API functions, and iii) a kernel module.

For the executable segments of a protected application, the gadget extraction toolkit identifies any instruction sequence ending in an indirect branch, starting from each and every byte of a segment. In the current version of our prototype we assume that the complete set of an application's modules is available in advance. However, it is possible to trivially relax this assumption by processing new modules on-the-fly at the time they are loaded by a protected application. The maximum gadget length is given as a parameter—in our experiments we conservatively used a length of 20 instructions. As discussed in Sec. 4.2.1, our extraction algorithm differs from previous approaches as it considers even instruction sequences that contain conditional or unconditional relative jumps. For this reason, code analysis explores all possible paths from every offset within a code segment, and follows recursively any conditional branches. The output of the analysis phase is two hash tables: one containing the offsets of `call`-preceded gadgets, and another containing the rest of the found gadgets. In the future, we will consider switching to Bloom filters to save space.

The overall operation of the runtime system is depicted in Fig. 4.8. The interposition component is implemented on top of the Detours framework [Hunt and Brubacher, 1999], which provides a library call interception mechanism for the Windows platform. During initialization, it requests by kBouncer's kernel module to enable the LBR feature on the CPU.

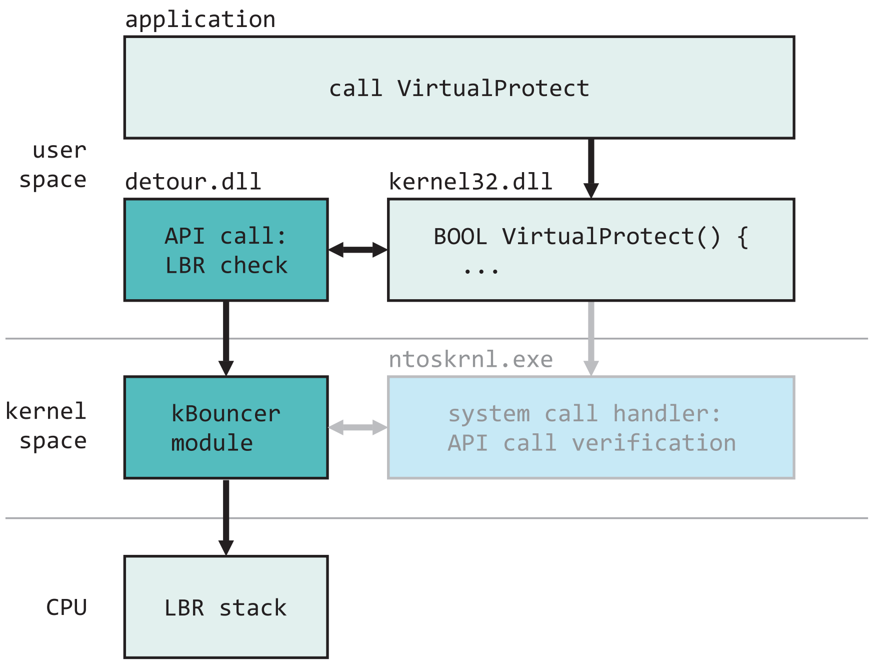


Figure 4.8: Overview of kBouncer's implementation. At the entry point of Windows API functions, kBouncer detours the execution, inspects the LBR stack in kernel mode, and then returns control back to the application.

The two components communicate through control messages over a pseudo-device that is exported by the kernel module (using the `DeviceIoControl` API function). Then, it selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component, instructing it to inspect the contents of the LBR stack for abnormal control transfers.

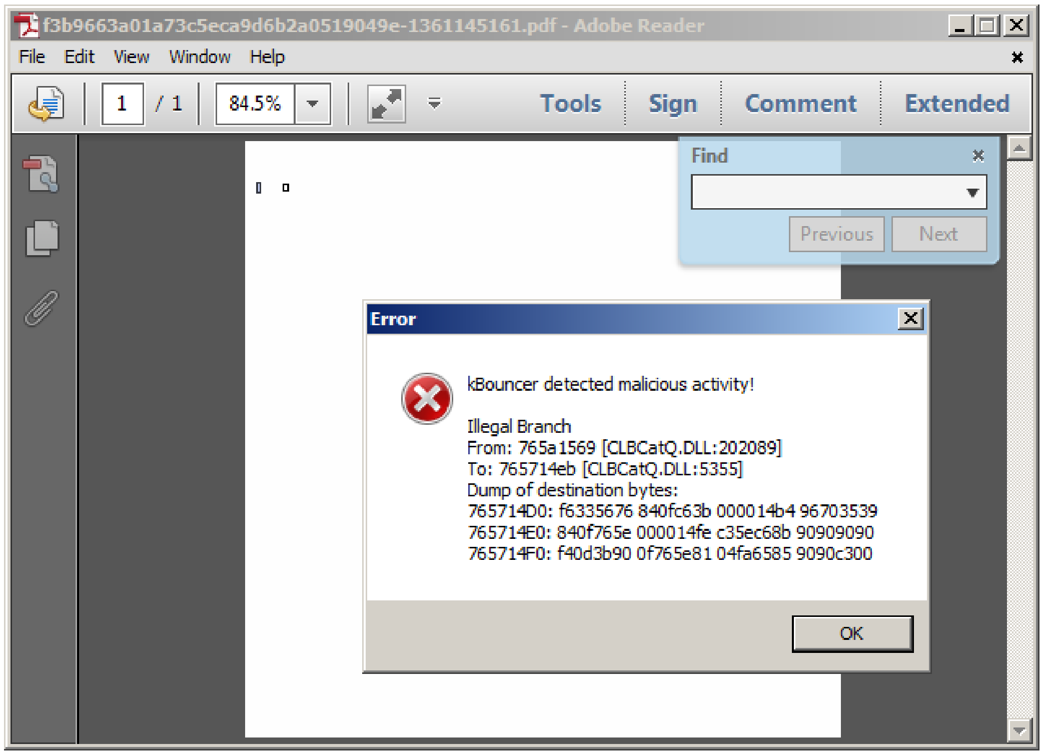
We note here that, althougt the use of Detours might seems to break the transparency feature of kBouncer, it is merely an implementation option. There are alternative ways to implement the same functionality, while still remaining completely transparent. For instance, by manipulating the page table permissions of the import address table, we could gain control each time a library call is performed.

The kernel module is responsible for three main tasks: i) enabling or disabling the LBR facility, ii) analyzing the recorded indirect branches, and iii) writing and verifying the appropriate checkpoint before allowing a system call to proceed. The first task involves reading and writing a few Model Specific Registers (MSR) using the `rdmsr` and `wrmsr` instructions. For the second task, whenever a control request is received from the user-space component, kBouncer analyzes the contents of the LBR stack, looking for the attributes described in Sec. 4.2. The MSR registers that hold the recorded information and configuration parameters are considered part of the running process context, and are preserved during context switches.

To identify illegal return instructions, the kernel module fetches a few bytes before each return target and attempts to decode any `call` instruction located right before the target instruction (call site check). Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any `ret` targets are looked up in the `call`-preceded gadgets hash table, whereas `call` or `jmp` targets are looked up in both hash tables, `call`-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function. To protect the kernel-level component from potential crashes when accessing invalid user-level locations, we use the `ProbeForRead` function of the Windows kernel API.

Unfortunately, the final task for API call verification has been only partly implemented, as it is not possible to perform system-call interposition in the current version of Windows 7. A recently added kernel feature in the 64-bit version of Windows, called PatchGuard [Field,], protects against kernel-level rootkits by preventing any changes to critical data structures, such as the System Service Descriptor Table (SSDT). Although this is effective against rootkits, PatchGuard removed the ability of legitimate applications, such as antivirus software, to intercept system calls. In response, Microsoft added a set of kernel-level APIs for filtering network and file-system operations (Windows Filtering Platform [Microsoft, f]). Hopefully, future OS versions will provide system call filtering capabilities as well.

Still, we did verify the correct operation of checkpoint verification by simulating it using the dataset of Table 4.2. We should note that this is not a design limitation, but only an implementation issue stemming from our choice of the target platform. For example, this would not have been an issue had we decided to implement kBouner for Linux, or any other open platform. For now, we plan to implement the checkpointing functionality for 32-bit


f3b9663a01a73c5eca9d6b2a0519049e-1361145161.pdf - Adobe Reader
File Edit View Window Help
1 / 1
84.5%
Tools
Sign
Comment
Extended
Find
Previous
Next
Error
OK

```
kBouncer detected malicious activity!

Illegal Branch
From: 765a1569 [CLBCatQ.DLL:202089]
To: 765714eb [CLBCatQ.DLL:5355]
Dump of destination bytes:
765714D0: f6335676 840fc63b 000014b4 96703539
765714E0: 840f765e 000014fe c35ec68b 90909090
765714F0: f40d3b90 0f765e81 04fa6585 9090c300
```

Figure 4.9: A screen capture of kBouncer in action, blocking a zero-day exploit against Adobe Reader XI.

applications by hooking system calls at user level through the WOW64 layer [Bremer, 2012] (which, however, will not provide the same protection guarantees as an actual kernel-level implementation).

In case an attack attempt is detected after the analysis of the recorded branches, the process is terminated and the user is informed with an alert message, as shown in Fig. 4.9. In this example, kBouncer blocks a malicious PDF sample that exploits an (at the time of writing) unpatched vulnerability in the latest version of Adobe Reader XI [Bennett *et al.*, 2013]. The displayed information, such as branch locations and targets, is supplied from the kernel-level module.

4.3.2 Analysis Framework

Moving from the basic concept to a functional prototype required a number of decisions that were mostly based on analyzing the behavior of large applications. To ease the effort required to perform this type of analysis, we developed an LBR analysis framework. Its goal is to provide a way to iterate over the LBR instances during the lifetime of an application, while at the same time providing useful information, such as translating addresses to function or image names. The framework is split in two parts: data gathering and analysis.

The data-gathering component is based on dynamic binary instrumentation. Although the runtime overhead of dynamic instrumentation is quite high (as discussed in Sec. 4.1.1), we use it here only for data gathering, which is an off-line and one-time operation. The tool we developed is built on top of Pin [Skaletsky *et al.*, 2010; Luk *et al.*, 2005], and records the following information during process execution: i) the file path and starting and ending address of any loaded executable image, ii) the location and name of any recognized function (e.g., exported functions), iii) the thread ID, location, and target of executed indirect branches (`ret`, `call` or `jmp`), iv) the thread ID, location, and number of system calls, and v) the thread ID, location, and return address of any identified function that was called.

The analysis part is a set of Python scripts that process the gathered data for each application. It provides a configurable LBR iterator which simulates different scenarios, such as returning LBR stack instances before system calls or certain function calls, or even after each branch is executed. To avoid mixing branches from different system threads in the same LBR instance, it internally keeps a list of separate LBRs per thread id. Finally, it provides convenient methods to translate addresses to function or image names when available.

4.4 Evaluation

In this section we present the results of our experimental evaluation of kBouncer in terms of runtime overhead and effectiveness against real-world ROP exploits. All experiments were performed on a computer with the following specifications: Intel i7 2600S CPU, 8GB RAM, 128GB SSD, 64-bit Windows 7 Professional SP1.

Table 4.3: Microbenchmarks.

Type	Number of Iterations	Avg. Total Time ms (stddev)	Avg. Single Time ns
HashLookup	1B	8231.6 (9.8)	8.2
IllegalRet	1B	10889.9 (312.9)	10.8
SysNull	10M	5145.0 (66.0)	514.5
SysLBR	10M	19981.8 (504.5)	1998.1
SysRead	10M	47267.7 (30925.6)	4726.7

4.4.1 Performance Overhead

4.4.1.1 Microbenchmarks

We started with some micro-benchmarks of different parts of kBouncer's functionality. Specifically, we measure the average time needed for the following operations, also listed in Table 4.3: hash table lookups ("HashLookup"), checks for illegal returns ("IllegalRet"), performing a system call ("SysNull"), reading the contents of the LBR stack ("SysLBR"), and reading parts of a process' address space ("SysRead").

In each case, we isolated the measured operation and tried to make the experiment as realistic as possible. For example, we extracted the hash table characteristics (domain size, hash table size, hit ratio) based on the dataset shown in Table 4.2. The data we used for the illegal return checks come from `kernel32.dll`, and use a worst-case workload by treating each location in its code segment as a possible return target. The next three experiments where measured in kernel level, as opposed to the first two. We measured the time needed to perform a no-op system call, a system call that only reads the LBR stack contents, and finally, a system call that in addition to reading the LBR stack, also fetches data from the sources and targets of each branch.

Table 4.3 shows the results of these benchmarks. Each benchmark runs the number of operations shown in the second column ten times, and calculates the average and standard deviation (next two columns). The last column shows the average time for a single operation. As we can see, looking up the hash table and checking for an illegal return are both very fast operations, in the order of a few nanoseconds. Performing a system call and reading the LBR stack are relatively more expensive, but still, in the order of a few microseconds. When attempting to access the instructions located at the source and target addresses of each branch record, the measured duration starts to fluctuate. We are not sure whether this behavior is normal, or it is a result of non-optimal use of the kernel API for accessing user-level memory. Overall, these microbenchmarks show that kBouncer's LBR stack analysis on each protected API function call takes on average no more than 5 microseconds.

4.4.1.2 Runtime Overhead

Measuring the performance overhead impact on interactive applications, such as web browsers and document viewers, is a challenging task. Instead, we decided to measure the performance overhead on programs that stress the core functionality of kBouncer, by making heavy use of the monitored Windows API functions. For this purpose, we used a subset of the tests provided in the test suite of Wine [Wine,], which repeatedly call Windows API functions with different arguments. To get more confident timing results, we kept only tests that do not interfere with external factors, such as network communication. The final set we used performs about 100,000 calls to Windows API functions that are protected by kBouncer, which is 20 times more than the protected calls made by the actual applications we previously tested (listed in Table 4.2).

Figure 4.10 shows the completion time for each of the different tests, with and without kBouncer. The average runtime overhead is 1%, with the maximum being 4% in the worst case. The total extra time spent across all tests when enabling kBouncer was 0.3 sec, a result consistent with the average cost of 5 μs per check based on our microbenchmarks (100,000 calls $\times$ 5 μs = 0.5 sec). Based on these results, which show that the performance overhead is negligible even for workloads that continuously trigger the core detection component, we believe that kBouncer is not likely to cause any noticeable impact on user experience.

4.4.2 Effectiveness

In the final part of our evaluation, we tested whether our prototype can effectively protect applications that are typically targeted by in-the-wild attacks, using the ROP exploits shown in Table 4.4. All exploits except the ones against Internet Explorer work on the

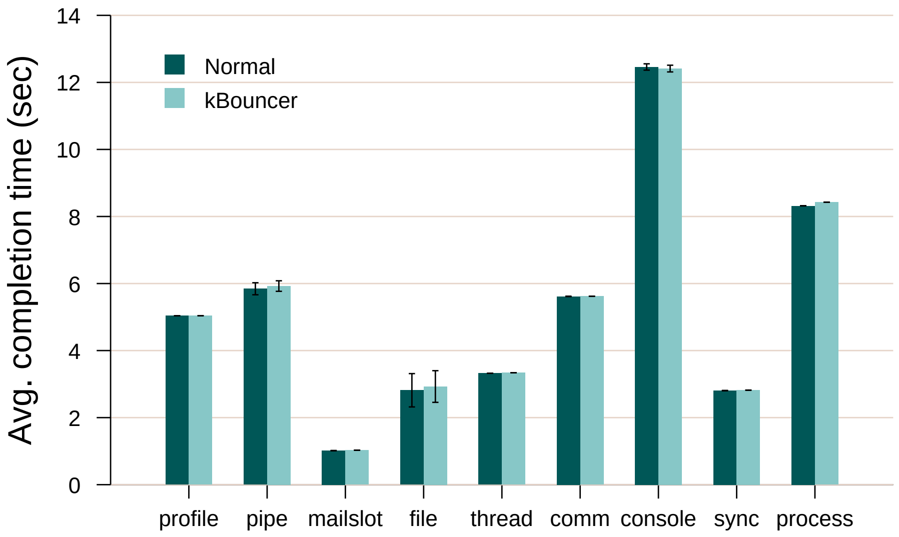


Figure 4.10: Execution time with and without kBouncer for Wine's `kernel32.dll` test suite, which resulted in the invocation of about 100K monitored Windows API functions. The average runtime overhead is 1%.

latest and up-to-date version of Windows 7 Professional SP1 64-bit. For the IE exploits to work, we had to uninstall the updates that fixed the relevant vulnerabilities (KB2744842 and KB2799329). We also had to tweak the ROP payload of the MPlayer exploit to correctly calculate the offset of `VirtualProtect` for the latest version of `kernel32.dll`, as the public version of the exploit was based on a previous version of that DLL.

The ROP code in the exploit against Adobe Reader v9.3.4 creates a file (`CreateFileA`), memory-maps the file in RWX mode (`CreateFileMappingA`, `MapViewOfFile`), copies the shellcode in the newly mapped area, and executes it. Similarly, the MPlayer and IE 8 exploits change the permissions of the memory region where the shellcode resides to RWX (`VirtualProtect`) and execute it. What is interesting about the IE 8 ROP code, is that it is constructed from the statically loaded Skype protocol handler DLL (`skype4com.dll`). The last two exploits in Table 4.4 were generated using the Metasploit Framework [Metasploit,]. For vulnerable applications that include widely used non-ASLR modules (like Java's `msvcrt71.dll`, which is loaded in Internet Explorer), Metasploit uses the same ROP payload based on `msvcrt71.dll`, which has been pre-generated by Mona [Corelan

Table 4.4: Tested ROP exploits.

Application	Vulnerability
Adobe Reader v11.0.1	Function pointer overwrite [Bennett *et al.*, 2013]
Adobe Reader v9.3.4	Stack-based overflow [Metasploit, 2010]
MPlayer Lite r33064	SEH pointer overwrite [Nate_M, 2011]
Internet Explorer 8	Use-after-free vulnerability [Parvez, 2013]
Internet Explorer 9	Use-after-free vulnerability [Metasploit, 2012b]
Adobe Flash 11.3.300	Integer overflow [Metasploit, 2012a]

Team, b]. This payload is similar to the one used in the MPlayer exploit, as it also uses `VirtualProtect` to bypass Data Execution Prevention (DEP). Finally, the Adobe Reader XI (v11.0.1) exploit is more complex, as it is the first in-the-wild exploit that uses ROP-only code, i.e., it does not carry any shellcode [Bennett *et al.*, 2013]. The malicious sample we tested ("Visaform Turkey.pdf") exploits a first vulnerability to escape from Reader's sandboxed process, and a second one to hijack the execution of its privileged process by loading a malicious DLL using `LoadLibraryW`.

In the first five exploits, the embedded shellcode simply invokes `calc.exe` using the `WinExec` Windows API call. The Reader XI exploit drops a malicious DLL. In all cases, we verified that the exploits worked properly on our testbed, by confirming that the calculator was successfully launched, or, for the Reader XI exploit, that the malicious DLL was loaded successfully. When kBouncer was enabled, it successfully blocked all exploits due to the identification of illegal returns at the time one of the `CreateFileA`, `VirtualProtect` or `LoadLibraryW` functions was invoked by the ROP code in each case.

4.5 Discussion

The Last Branch Recording feature of recent Intel processors is what enables kBouncer to achieve its transparent and low-overhead operation. Many of our design decisions are corollaries of the very limited size of the LBR stack, which in the most recent processors holds only 16 records. Given that previous processor generations had even more size-constrained LBR implementations, this is definitely a significant improvement, and hopefully future processors will support even larger LBR stacks. This would allow kBouncer to achieve even higher accuracy by inspecting longer execution paths, making potential evasion attempts even harder.

Currently, an attacker could evade kBouncer by ensuring that the final 16 executed gadgets before the invocation of an API function are considered legitimate. Specifically, given that kBouncer looks for both illegal returns and gadget chaining in parallel, this would require i) all 16 gadgets to be either `call`-preceded or non-`ret` gadgets, and ii) at least one out of every eight of them (eight is our current gadget chaining detection threshold) to be longer than 20 instructions.

Recent work confirmed that manually constructing such ROP payloads is possible in some cases [Carlini and Wagner, 2014; Schuster *et al.*, 2014; Göktaş *et al.*, 2014b; Davi *et al.*, 2014]. However, a more thorough analysis is still missing and it is not clear whether it is possible to automate the construction of these payloads as the available gadgets are significantly more complex. Our preliminary evidence (Section 4.2.1), shows that only 6.4% of all gadgets ending with `ret` are `call`-preceded, and this is when considering even fragmented gadgets up to 20 instructions long (this percentage drops to 3% when considering gadgets with at most five instructions). The number of `call`-preceded can be further decreased by putting additional restrictions on the targets of return instructions (part of our future work — see Section 6.2). In addition, ROP compilers like Q [Schwartz *et al.*, 2011] typically take into account non-fragmented gadgets up to five instructions long. Longer gadgets incur more CPU state changes, which complicate the (either manual or automated) gadget arrangement process. Indicatively, for a similar set of applications, even when 20% of all gadgets are available, Q could not generate a functional payload [Pappas *et al.*, 2012]. Note that the selection of a maximum gadget length of 20 instructions was arbitrary—four times the typically used standard seemed enough. If evasion becomes an issue, longer gadgets could be considered during the gadget chaining analysis of an LBR snapshot.

Alternatively, an attacker could look for a long-enough execution path that leads to the desired API call as part of the application's logic. Such a path should satisfy the following constraints: (i) contain at least 16 indirect branches, the targets of which happen to lead to the execution of the desired API function, and (ii) the executed code along the path should not alter the state or the function arguments set by the previously executed ROP code. Finding such a path seems quite challenging, as in many cases the desired function might not be imported at all, and the path should end up with the appropriate register values and arguments to properly invoke the function. This is even more difficult in 64-bit systems, where the first four parameters are passed trough registers, as opposed to the 32-bit standard calling conventions in which parameters are passed through the stack.

A way to extend kBouncer's view of the execution paths that lead to API function calls would be to dynamically insert additional inspection hooks at previous locations along a path. At runtime, the first time a particular API function is called, the system can "walk" back on the execution path that led to the call, up to 16 indirect branches away, and insert a detection hook (without any checkpointing) at the farthest appropriate place. The next time the flow of control goes through the same path, the ROP code check will also be triggered at an earlier point. The same process can continue on future invocations in a "push-back" way, until a long enough path has been covered. New checks can be inserted using a dynamic binary instrumentation framework, like Pin [Luk *et al.*, 2005]. The number of instrumented points should only be a small fraction of the executed instructions, so the performance overhead is expected to remain low. The path distance can be selected to allow for an acceptable trade-off between detection depth and potential increases in the runtime overhead due to the larger number of hooks. Apart from protecting against long execution paths to erase branch history, the push-back extension could be used to validate known execution paths — assuming checkpoints are added during a training phase. In this case, even if an attacker tries to use long-enough or allowed gadgets, the execution path will probably be flagged as unknown. Another, more promising, approach for preventing the use of long gadgets is changing the gadget chain length to count chains based on the usefulness of the gadgets, rather than length alone. For instance, a long gadget that does not clobber any register is probably more useful that a shorter one that clobbers a few. Computed memory accesses is another factor that can limit the usefulness of a gadget. Exploring this direction is part of our future work (see Section 6.2).

Our selection of sensitive Windows API functions was made empirically based on a large set of different shellcode and ROP payload implementations [Metasploit, ; Nepenthes, 2007;

Polychronakis *et al.*, 2009; Immunity, 2010; Corelan Team, b; Schwartz *et al.*, 2011]. A list of the 52 currently protected functions is provided in the appendix. Although current ROP exploits rely mainly on only a handful of API functions (see Sec. 4.4.2), we have included many others that have been used in the past in legacy shellcode, as some exploits might implement their whole functionality using purely ROP code (as demonstrated recently by an exploit against the latest version of Adobe Reader XI [Bennett *et al.*, 2013]). The set of protected functions can be easily extended with any additional potentially sensitive functions that we might have left out. Although it would be possible to protect all Windows API calls, we believe that this would not offer any additional protection benefits, and would just introduce unnecessary overhead.

4.6 Combining with In-place Code Randomization

In this section, we explore the benefits of combining in-place code randomization with indirect branch tracing. As stated before, these techniques are both compatible and complementary to each other.

The biggest advantage of combining the techniques is in the protection coverage. Most of the proposed ROP defenses fall under either of these two categories: (i) break the knowledge of the code layout, or, (ii) restrict the use of indirect branches (see Section 2.2). Specifically to our work, in-place code randomization falls under the first, whereas indirect branch tracing falls under the second. Thus, combining them leads to a more complete solution against ROP. On the other hand, the combination of the two techniques will inevitably lack one of indirect branch tracing's features: transparency. Still, giving up transparency for a more complete solution is arguably justifiable under many scenarios.

There are two ways in which the techniques complement each other. First, indirect branch tracing can prevent ROP attacks from using non-randomized gadgets. As shown in Section 3.3, in-place code randomization breaks 80% of the gadgets, on average. Although we also showed that the remaining gadgets in our dataset are not enough for automated construction toolkits, there is still the possibility that a determined attacker could manually construct a ROP payload. This is where indirect branch tracing could help by detecting

Table 4.5: In-place code randomization coverage on gadgets, found in extracted code, ranging from one to five instructions compared to twenty up to fifty instructions. Coverage improves by 20% for longer gadgets.

	1–5 Instr. Gadgets		20–50 Instr. Gadgets	
Software	Total	Modifiable (%)	Total	Modifiable (%)
Adobe Reader 9	1,207K	943K (78.1)	101K	99K (98.1)
Firefox 4	455K	381K (83.7)	46K	45K (98.7)
iTunes 10	373K	293K (78.5)	43K	42K (97.4)
Windows XP SP3	7,897K	6,452K (81.7)	636K	627K (98.5)
Windows 7 SP1	15,703K	12,970K (82.6)	1,583K	1,551K (98.0)
Total	25,636K	21,041K (**82.1**)	2,412K	2,366K (**98.1**)

ROP attacks using static gadgets. Second, in-place code randomization can break long gadgets more easily, thus preventing an attacker for using them to evade the gadget chain length check of indirect branch tracing. Using long gadgets is already much harder because of their side-effects, and severely limiting their number might make it impossible in many cases. In the current evaluation of in-place code randomization, we set the maximum gadget length to five instructions. This limit does not only reflect the reality better, but stresses in-place code randomization as is it harder to randomize smaller code fragments.

Intuitively, we expect the gadget randomization coverage to improve for longer gadgets, as the probability of randomizing some bytes within them is greater. We re-ran the coverage evaluation using the same dataset (detailed in Table 3.1), while increasing the gadget length to the range of twenty to fifty instructions, instead of one to five. Table 4.5 compares the results for longer gadgets, which basically confirms our intuition. There are two pairs of columns with raw numbers for the total gadgets within the extracted code and the modifiable ones for the two gadget length ranges we considered. The drop in the second pair is due to the fact that is it less frequent to find long instruction sequences without control flow transfers. Overall, randomization coverage, in terms of modifiable gadgets, increased from 82.1% to 98.1%. This clearly demonstrates the robustness of in-place code randomization against loner gadgets, and, moreover, the potential for combining the techniques.

Chapter 5

Dynamic Relocation Reconstruction

Keeping systems up-to-date with the latest patches, updates, and operating system versions, is a good practice for eliminating the threat of exploits that rely on previously disclosed vulnerabilities. Major updates or newer versions of operating systems and applications also typically come with additional or improved security protection and exploit mitigation technologies, such as the stack buffer overrun detection (/GS), data execution prevention (DEP), address space layout randomization (ASLR), and many other protections of Windows [Miller *et al.*, 2011], which help in defending against future exploits.

At the same time, however, updates and patches often result in compatibility issues, reliability problems, and rising deployment costs. Administrators are usually reluctant to roll out new patches and updates before conducting extensive testing and cost-benefit analysis [Rescorla, 2003], while old, legacy applications may simply not be compatible with newer OS versions. It is indicative that although Windows XP SP3 went out of support on April 8th, 2014 [Microsoft, g], many home users, organizations, and systems still rely on it, including the majority of ATMs [Summers, 2014]. In fact, the UK and Dutch governments we forced to negotiate support for Windows XP past the cutoff date, to allow public-sector organizations to continue receiving critical security updates for one more year [UKX, 2014].

As a step towards enhancing the security of legacy programs and operating systems that do not support the most recent exploit mitigation technologies, application hardening tools such as Microsoft's EMET (Enhanced Mitigation Experience Toolkit) [Microsoft, a] can be used to retrofit these and even newer (sometimes more experimental) protections on third-party legacy applications. An important such protection is address space layout randomization, which aims to defend against exploitation techniques based on code reuse, such as return-to-libc [Designer, 1997] and return-oriented programming (ROP) [Shacham, 2007].

ASLR randomizes the load address of executables and DLLs to prevent attackers from using data or code residing at predictable locations. In Windows, though, this is only possible for binaries that have been compiled with *relocation* information. In contrast to Linux shared libraries and PIC executables, which contain position-independent code and can be easily loaded at arbitrary locations, Windows portable executable (PE) files contain absolute addresses, e.g., immediate instruction operands or initialized data pointers, that are valid only if an executable has been loaded at its preferred base address. If the actual load address is different, e.g., because another DLL is already loaded at the preferred address or due to ASLR, the loader adjusts all fixed addresses appropriately based on the relocation information included in the binary.

Unfortunately, PE files that do not carry relocation information cannot be loaded at any address other than their preferred base address, which is specified at link time. Relocation information is often stripped from release builds, especially in legacy applications, to save space or hinder reverse engineering. Furthermore, in 32-bit Windows, it is not mandatory for EXE files to carry relocation information, as they are loaded first, and thus their preferred base address is always available in the virtual address space of the newly created process. For these reasons, tools like EMET unavoidably fail to enforce ASLR for executables with stripped relocation information. Consequently, applications with stripped relocation information may remain vulnerable to code reuse attacks, as DEP alone can protect only against code injection attacks. Furthermore, recently proposed protection mechanisms for Windows applications rely on accurate code disassembly, which depends on the availability of relocation information, to apply control flow integrity [Zhang *et al.*, 2013] or code randomization [Pappas *et al.*, 2012].

In this work, we present a technique for reconstructing the missing relocation information from stripped binaries, and enabling safe address space layout randomization for executables which are currently incompatible with forced ASLR. The technique is based on discovering at runtime any stale absolute addresses that need to be modified according to the newly chosen load address, and applying the necessary fixups, replicating in essence the work that the loader would perform if relocation information were present. Again, as transparency is a key requirement for the practical applicability of protections tailored to third-party applications (see Section 1.2), the proposed approach relies only on existing operating system facilities (mainly page table manipulation) to monitor and intercept memory accesses to locations that need fixup.

We have evaluated the performance and effectiveness of our prototype implementation using the SPEC benchmark suite, as well as several Windows applications. Based on our results, incremental runtime relocation patching is practical, incurs modest runtime overhead for initial runs of protected programs, and has negligible overhead on subsequent runs, as the reconstructed relocation information is preserved. Besides forced ASLR, the proposed technique can also be used to resolve conflicts between stripped binaries with overlapping load addresses, a problem that occasionally occurs when running legacy applications, and to significantly improve code disassembly. Finally, the use of relocation information improves the accuracy of disassembling binary code, which in turn enables or improves other protection techniques that rely on it [Pappas *et al.*, 2012; Hiser *et al.*, 2012; Wartell *et al.*, 2012; Zhang *et al.*, 2013; Zhang and Sekar, 2013].

5.1 Relocation Information in Windows

In Windows, which is the main focus of this work, ASLR support was introduced in Windows Vista. By default, it is enabled only for core operating system binaries and programs that have been configured to use it through the `/DYNAMICBASE` linker switch. For legacy applications, not compiled with ASLR support and other protection features, Microsoft has released the Enhanced Mitigation Experience Toolkit (EMET) [Microsoft, a], which can be used to retrofit ASLR and other exploit mitigation technologies on third-party applications. A core feature of EMET is Mandatory ASLR, which randomizes the load address of modules even if they have not been compiled with the `/DYNAMICBASE` switch, but do include relocation information. This is particularly important for applications that even though have opted for ASLR, may include some DLLs that remain in static locations, which are often enough for mounting code reuse attacks [Fresi Roglia *et al.*, 2009; Zovi, 2010b; Johnson, 2011]. EMET's ASLR implementation also provides higher randomization entropy through additional small memory allocations at the beginning of a module's base address. Many of the advanced ASLR features of EMET have been incorporated as native functionality in Windows 8, including forced ASLR.

The above recent developments, however, are not always applicable on legacy executables. Typically, when creating a PE file, the linker assumes that it will be loaded to a specific memory location, known as its *preferred base address*. To support loading of modules at addresses other than their preferred base address, PE files may contain a special `.reloc` section, which contains a list of offsets (relative to each PE section) known as "fixups" [Skape, 2007]. The `.reloc` section contains a fixup for each absolute addresses at which a delta value needs to be added to maintain the correctness of the code in case the actual load address is different [Pietrek, 2002]. Although DLLs typically contain relocation information, release builds of legacy applications often strip `.reloc` sections to save space or hinder reverse engineering. This can be achieved by providing the `/FIXED` switch at link time. Furthermore, in older versions of Visual Studio, the linker by default omits relocation information for EXEs when performing release builds, as the main executable is the first module to be loaded into the virtual address space, and thus its preferred base address is always expected to be available.

As modules (either EXEs or DLLs) with stripped relocation information cannot be loaded at arbitrary addresses, the OS or tools like EMET cannot protect them using ASLR. Legacy applications may also occasionally encounter address conflicts due to different modules that attempt to use the same preferred base address. Our system aims to enable the randomization of the load address of modules with stripped relocation information by incrementally adjusting stale absolute addresses at runtime.

5.2 Approach

Our approach to the problem of relocating stripped binaries relies on reconstructing the missing relocation info by discovering such relocatable offsets at runtime. We note here that a static approach, i.e., using disassembly to find all the relocatable offsets, would be much more difficult, if not infeasible in many cases—the reason being that stripped binaries also lack debugging symbols, so complete disassembly coverage would be impossible in most cases.

5.2.1 Overview

The basic idea of our approach is to load the stripped binary at a random location and monitor any data accesses or control transfers to its original location. Any such access to the original location is either a result of using a relocatable offset or an attack attempt (the attacker might try to reuse parts of the original code, not knowing that the binary was relocated). The next step is to identify the source of the access by checking whether it was indeed caused by a relocatable offset. In this case, the offset it located, its value is fixed to the new random base, and the relocation info is reconstructed so as next time the same program is executed a fixup for that address can be automatically applied.

Although there are a few different ways to monitor memory access and control transfers at runtime, we followed an approach that minimizes its effects and dependencies on third-party components. For instance, instruction-level dynamic binary instrumentation was not considered for this reason, as it requires the installation of third-party dynamic binary instrumentation frameworks (and typically incurs a prohibitively high runtime overhead). Our monitoring facility is built around basic operating system functionality, mostly memory protection mechanisms. More precisely, after a binary is loaded to a random location, we change the permissions of its original location to inaccessible, so as each time a memory access or control transfer happens to one of the original locations, a memory violation exception is raised. This type of exception usually contains the location of the instruction that caused it, the faulting address (can be the same as the instruction location), and the type of access (read or write).

The main challenge of our approach now becomes to identify whether an access to the original binary location is caused by a relocatable offset and how to trace it back to that offset. To better explain this issue, consider the following example. Assume that an instruction updates the contents of a global variable using its absolute address (e.g., `0x1000`). When the instruction is executed from the new, randomly chosen location of the binary, an exception will be raised. At this point, we know the location of the instruction and the faulting address (`0x1000`). After analyzing the faulting instruction, we see that one of its operands is actually the faulting address. In this case, we have to fix the operand by adjusting it to the new random base, and also reconstruct the relocation info of this offset.

The example above is the most straightforward case of identifying a relocatable offset. In practice, in most cases the relocatable offset is not part of the faulting instruction. For example, consider the case of dereferencing a global pointer. There is an instruction to load the value of the pointer, probably in a register, and another instruction to read the contents of the memory location stored in the register. In this case, the faulting address is not directly related with the faulting instruction. Even worse, there are cases in which the relocatable offset has been changed before it is used. For example, accessing a field from a structure in a global array would only require a single relocatable address (the location of the array) and would result in many runtime accesses within the range of the array. It is very difficult to trace such an access reliably back to its source relocatable offset.

However, code-reuse attacks rely solely on the knowledge of the code's location, regardless of the location of data. Based on this observation, and due to the problematic nature of data pointer tracing, we focus on randomizing the load address of code segments only. Code pointers are usually guaranteed not to support any arithmetic—it would be difficult to imagine code that depends on expressions such as adding a few bytes to the location of a function start, at least for compiler-generated code. An exception to this is jump tables that contain relative offsets, but this is a case that can be easily covered, as we will see later on. This simplifies the overall approach, without sacrificing any of the security guarantees.

Figure 5.1 shows a high-level overview of our approach. When a stripped binary is loaded for execution (left side), its code segment is moved to a random location, while the

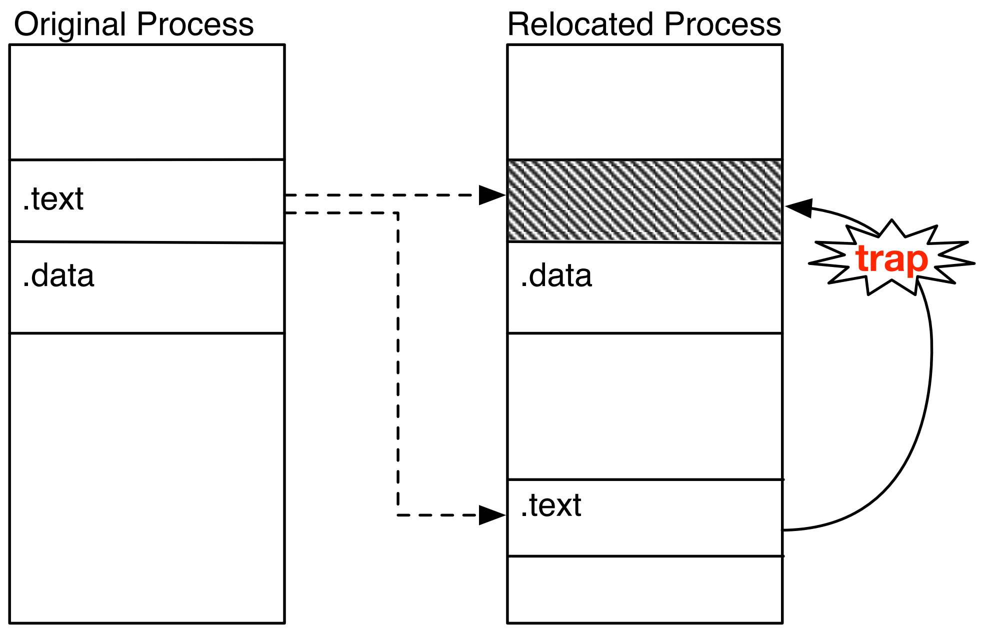


Figure 5.1: High-level overview of runtime relocation fixup. The code segment of a stripped binary is loaded to a randomly chosen location, and its original memory area is marked as inaccessible. Memory accesses and control transfers to any of the original locations are trapped. Relocation information is then reconstructed by analyzing the faulting instruction.

original location becomes inaccessible (right side). Then, whenever there is a memory access or control transfer to the original location (solid arrow), the faulting address along with the instruction that caused it are analyzed. Based on this analysis, the source relocatable offset is pinpointed, gets fixed, and its relocation information is reconstructed. In the following, we describe in more detail how this analysis is being performed.

5.2.2 Access Analysis

The series of steps performed after a memory access violation exception is raised due to a memory access in the original code location is depicted in Figure 5.2. Broadly speaking, access violations are grouped into two categories based on their root cause: (i) reading the contents of the original code segment, and (ii) control transfers to the original code segment. To distinguish between the two, the system checks whether the value of the instruction pointer is within the original code segment.

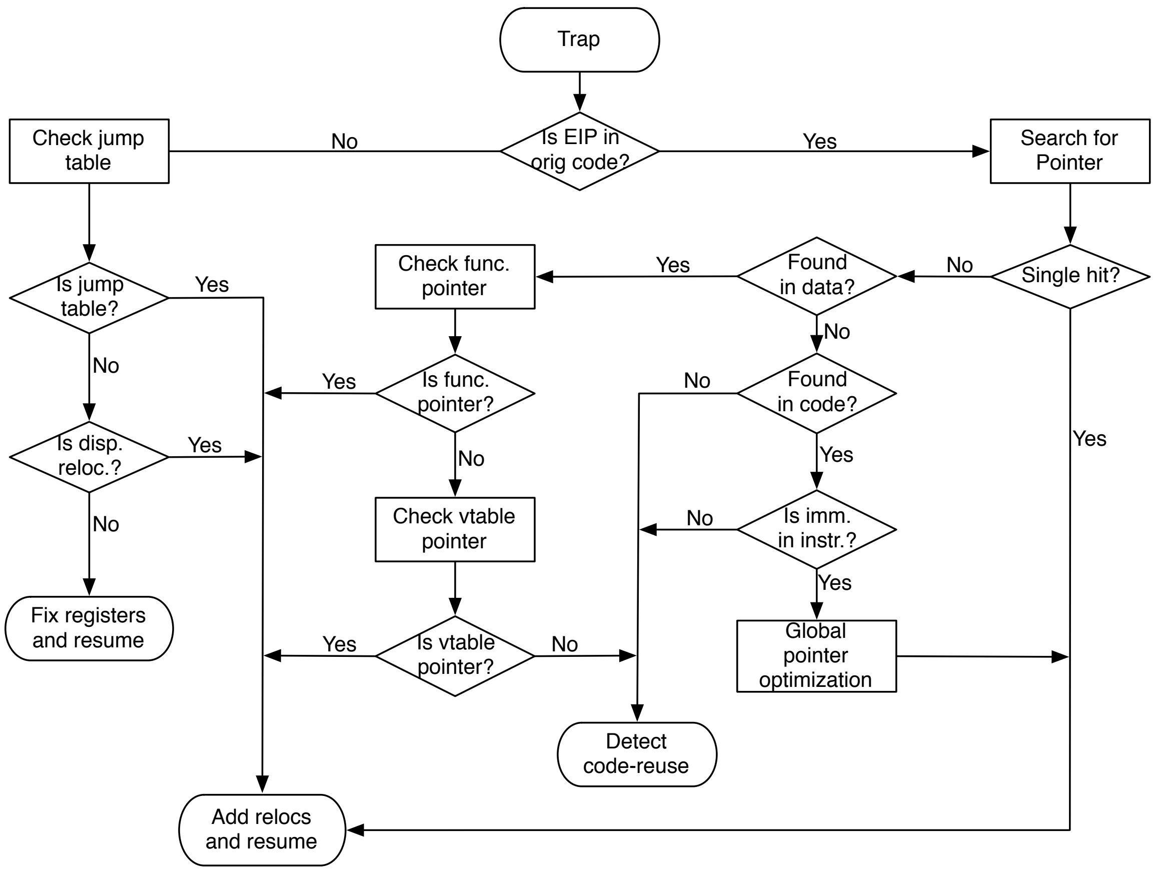


Figure 5.2: Flow graph of the procedure followed after a memory access exception (trap) is generated. If the instruction pointer (EIP register) at the time of the exception is within the original code segment, the system performs pointer verification, otherwise the faulting instruction is fixed.

In practice, the first case corresponds mostly to indirect jump instructions that read their target from the code segment. These are typically part of jump tables, which are used for the implementation of switch statements in C. In the second case, control is transfered to the original code segment because a code pointer that has not been relocated is used. This could be a simple function pointer, part of a C++ virtual table (vtable), or a static one, represented as an immediate value in an instruction. In the following subsections we describe in detail how each of these cases is handled.

When control is transferred to locations in the original code segments for which there is no code pointer, or when we can not verify it as a legitimate code pointer, these transfers are flagged as code-reuse attempts (see Fig. 5.2). This effectively allows attackers to reuse code paths for which there are legitimate code pointers (e.g., function entries or jump table targets), given that they have not been reconstructed yet. Arguably, this leaves a very limited set of gadgets for the attacker, which quickly shrinks further as relocatable code pointers are identified.

5.2.3 Jump Tables

A jump table is an array of code targets that is usually accessed through an indirect jump. The following is an example of such a jump table in x86 assembly (taken from `gcc`'s binary):

```
.text:004D5CCE    jmp  ds:off_4D6864[eax*4] ; switch jump
...
; DATA XREF: _main+2CE ; jump table for switch statement
.text:004D6864 off_4D6864    dd offset loc_4D5D53
.text:004D6868               dd offset loc_4D5D63
.text:004D686C               dd offset loc_4D5D93
.text:004D6870               dd offset loc_4D5D8B
```

When the `jmp` instruction is executed from the new random location, an exception is going to be raised, with the faulting address being `(0x4D6864 + eax * 4)`. This is handled as follows: i) starting from the location pointed to by the faulting address, we scan the bytes before and after that location for more addresses and fix them, and ii) we also fix the relocatable offset in the address operand of the indirect jump instruction. In case of jump tables with relative offsets, we just skip the first step.

5.2.4 Pointer Verification

After jump tables are covered, we only expect to see control flow transfers to the locations of the original code. In these cases, the location of the faulting instruction is also the faulting address—there is no information about the source instruction. Given a faulting address, the whole code segment and initialized data are scanned for all its occurrences. If there is a single occurrence, we assume that it is a relocatable offset, which is handled appropriately.

Otherwise, for each occurrence in the code segment, we verify that it is indeed part of a valid instruction—more precisely, an immediate operand.

Occurrences found in the initialized data segments are a bit more complicate to cover. Usually, for such a hit to be indeed a relocatable offset, it has to be a variable holding a function pointer, so there should be a way of accessing that variable. To verify this, we just need to find a data reference to that variable. In addition, function pointers can be parts of structures, arrays, or a combination of both. In general, we verify that an occurrence of the faulting address in the data segment is a relocatable offset that needs to be fixed if we can find a reference to or near its location (given as a parameter).

The following example illustrates the function pointer verification process. Assume there is a global variable that is statically initialized with the address of a function. Also, there is an indirect call instruction that reads the value of the global variable and transfers control to its value. At runtime, the value is going to be read (because the data segment is not relocated) and an exception is going to be raised when control is transfered to the function. Both the faulting address and the faulting instruction will correspond the beginning of the target function. At this point, we find an occurrence in the code segment and verify that it belongs to an instruction—which is the indirect call in this case.

Another use of function pointers is in C++ virtual tables, which is how dynamic class methods are represented. These pointers are handled a bit differently than simple function pointers, and, for this reason, we have introduced special checking rules. We first verify that there is a move instruction that copies the head of the table to a newly created class instance, by finding a move instruction that references a memory location close to the place where the code pointer was found. We then also verify that the control was transferred by an indirect call through a register, by reading the current value at the top of the runtime stack (return address) and disassembling the instruction right before the location it points to. Bellow is a real example taken from the `eon` binary of the SPEC benchmarks suite:

```
;; function call
.text:004017F9    mov   eax, [ecx]    ; ecx is this ptr
.text:004017FB    mov   eax, [eax+24h]
.text:004017FE    push  edx
```

```
.text:004017FF    mov   edx, [ebp+arg_4]
.text:00401802    push  edx
.text:00401803    mov   edx, [ebp+arg_0]
.text:00401806    push  edx
.text:00401807    call  eax
.....
;; vtable (the static part)
; DATA XREF: sub_409B40+8o  ; sub_40B0E0+2Fo
.rdata:00461D24 off_461D24    dd offset sub_40AAD0
.rdata:00461D28               dd offset sub_409BB0
.rdata:00461D2C               dd offset sub_409BC0
.....
;; copying the head of the table
.text:0040B10C    lea   ecx, [esi+4]    ; this
.text:0040B10F    mov   dword ptr [esi], offset off_461D24
```

The top part of the example shows the code that loads the function pointer from the vtable to the `eax` register and then transfers control there by calling it. The call instruction at the end will actually going to raise an exception. While handling the exception, we check (i) the table that contains the faulting address at `0x461D24` (middle part) is referenced by a move instruction at `0x40B10F` (bottom part), and (ii) the instruction before the return address is a call instruction with a register operand (at `0x401807`).

5.2.5 Dynamic Data

Although in order to reconstruct the missing relocation information we need to locate relocatable offsets within the image of the executable module, copies of such values also appear in dynamic data (e.g., in the stack or heap). This is the result, for example, of a global pointer being copied in a structure field that was dynamically allocated. In this case, an exception is going to be raised when the copy of the pointer (in the structure) is used. As described before, our technique is going to trace the original relocatable offset. This is sufficient for reconstructing the relocation information for this pointer, and avoid dealing with the same problem next time the same program is executed. However, we do not take any further actions to deal with copies in dynamic data. Thus, we might have to handle more than one exceptions for the same relocatable value during the same run in which it was first discovered. This, of course, does not affect the correctness and robustness of the technique in any way, but can affect overall performance.

To avoid the performance penalty under some cases, while not weakening our original approach, we added a simple optimization for global pointers. Each time a relocatable offset is fixed, and it is found to be the source operand of an instruction that copies it over to a global data location, we check whether the destination memory location contains the same value and relocate that copy, too. Below is an example of a few such instructions (taken from `gcc`'s binary):

```
.text:004D5A69    mov   dword_550968, offset loc_4D1F10
.text:004D5A73    mov   dword_550AAC, offset loc_4D1C20
.text:004D5A7D    mov   dword_5509C4, offset nullsub_1
```

The first `mov` instruction in the above example copies the (relocatable) offset `loc_4D1F10` to the global data memory location `0x550968`. At the time an exception is raised because control was transfered to address `0x4D1F10`, the source operand of the first `mov` instruction will be fixed, and, if the same value is found at address `0x550968`, that will be fixed as well. In this way, future copies of the relocatable offset will point to the new code location, and no more exceptions will be raised for this instance.

In general, when this optimization is not applicable and there are many copies of relocatable offsets being repeatedly used, we have the option to set an access threshold, beyond which the system can inform the user that restarting the program would greatly increase its performance. Still, we believe that this is a minor issue, as it might occur only in the first few times a program is executed. After that, the relocation information of the majority of the relocatable offsets will have been reconstructed.

5.3 Implementation

We built a prototype of the described technique for the Windows platform. Most of the development of the tool was done on Windows XP. However, as the APIs we use have not changed in more recent versions of the operating system, our prototype supports even the latest version, which is Windows 8.1 at the time of writing.

The most significant part of the implementation is built on top of the Windows Debugging API [Microsoft, e], with the addition of some other standard functions (e.g., `CreateProcess`). This API is designed to work between two processes: the parent process is responsible for spawning a child process, and then capture and analyze any debug events the child generates. Debug events include memory access violation exceptions, process/thread startup/termination, and so on. Our implementation is bundled as a single application (about 1.5 KLOC) which can be executed from the command prompt, and receives the path of the target program to be protected as a command-line argument.

At a higher level, there are two phases of operation: initialization and runtime. We discuss both in sufficient detail in the rest of this section.

5.3.1 Initialization

The first step during the initialization phase is to spawn the process, while passing the appropriate arguments in order to enable debugging. The very first debug event generated by the child process is a process creation event, which is handled by the parent by performing the following tasks before resuming the execution of the child process. Initially, the Portable Executable (PE) headers are parsed. These headers include information such as the boundaries of each section (data, code, etc.) and the entry point of the code. Given that information, we proceed by copying the code section to a new, randomly chosen location using the `ReadProcessMemory` and `WriteProcessMemory` API functions, while changing the memory protections of the original code segment to inaccessible using `VirtualProtectEx`.

In order to improve the performance of certain runtime operations, a hash table of all possible code pointer values is built. This is done by scanning all sections and inserting any four-byte values (assuming 32-bit processes) that fall into the address range of the original code segment. Finally, we check whether there is a file that contains relocation information that was discovered as part of previous runs, and apply them.

5.3.2 Runtime

After initialization is completed and control is given back to the child process, the parent blocks while waiting for the next debugging event. Usually, we expect memory access violation exceptions to be generated after this stage. New DLL loaded events might happen as well, but rarely. Whenever a new DLL is loaded in the address space of the child process, the system checks whether it contains relocations. In case it does not, the same initialization steps that were previously described are performed.

As described in Section 5.2, the core of our technique is implemented as part of the handling mechanism of memory access violation exceptions. Each exception record contains information about the location of the instruction that caused it, along with the faulting address. Based on this information, we distinguish between two main cases: i) the instruction pointer falls within the address range of the original (inaccessible) code segment (instruction address and faulting address are the same), and ii) an illegal memory access was made by an instruction located in the relocated code segment (instruction address and faulting address are different).

If the instruction pointer after a memory exception is received falls within the original code segment, this means that the control flow was transfered there and the program failed when it tried to execute the next instruction. In this case, the faulting address corresponds to the location of the instruction in the exception record. The exception is handled by first looking up the faulting address in the hash table—which is constructed during the initialization phase. A single hit is the simplest case, because it means that this is the source of the exception. If there are more than one hits, each one is verified using the rules described in Section 5.2 for immediate values or function pointers.

Alternatively, if the faulting instruction belongs to the relocated code segment, this means that one of its operands caused the fault. This happens under two circumstances: the instruction is an indirect jump, reading a jump table target from the original code location, or an instruction that uses a copy of a relocatable value from dynamic data.

5.4 Evaluation

In this section we present the results of the experimental evaluation of our prototype in terms of correctness and performance overhead. For the largest part of our evaluation, we used benchmarks from SPEC CPU2006 [SPEC, 2006], as well as some real-world applications, such as Internet Explorer and Adobe Reader. All the experiments were performed on a computer with the following specifications: Intel Core i7 2.00GHz CPU, 8GB RAM, 256GB SSD with 64-bit Windows 8.1 Pro.

5.4.1 Statistics

We started our evaluation with the goal of getting a better feeling on the differences of applying our technique to programs with distinct characteristics. First, we selected all the test programs in the integer suite that come with the SPEC benchmark and stripped the relocation information from the compiled binaries. Out of the twelve programs in that set, only `libquantum` had to be left out because it uses some C99 features that are not supported by Visual C++ (as noted in the SPEC configuration file Example-windows-ia32-visualstudio.cfg). Then, we executed each one using our prototype and gathered some valuable statistics that provide insights about the runtime behaviour of our technique. At the same time, we checked that the output of the benchmark test runs was correct, which in turn verified the correctness of our implementation under these cases.

Table 5.1 shows the results of this run. The first column contains the name of each SPEC test program, followed by the number of possible pointers that we identified for each during the initialization phase. The next three columns show the number of identified jump tables and the number of verified pointers along with the percentage of them that had a single hit in the possible pointers set. Next, we have the number of times that an already fixed relocatable offset reappeared at runtime because of copies of it in dynamic data, followed by the number of global pointer copies that we were able to apply the optimization described in the last part of Section 5.2. Finally, the number of actual relocatable offsets that we were

Table 5.1: Statistics from running the SPEC benchmarks using the reference input data (largest dataset).

Program	Possible Pointers	Jump Tables	Verified Pointers	Single Hit	Dynamic Data	Global Opt.	Reconst. Reloc.
perlbench	31,260	118	633	83.0%	43M	41	2,614
bzip2	2,147	4	11	84.6%	25	4	76
gcc	98,955	510	1,008	65.2%	73M	269	7,849
mcf	1,875	1	13	100.0%	19	-	22
gobmk	69,852	21	968	63.5%	4M	54	1,270
hmmer	4,798	15	17	94.4%	42	2	152
sjeng	8,460	12	17	100.0%	18	-	135
h264ref	17,526	17	27	71.0%	320K	61	209
omnetpp	24,861	13	1,509	90.6%	269K	8	1,669
astar	2,690	2	20	100.0%	31	-	42
xalancbmk	141,246	54	4,402	84.2%	9M	24	5,392

able to reconstruct their relocation information in shown in the last column.

An interesting observation is that most of the times we have a single hit during the verification of a code pointer, which simplifies the overall procedure. Another interesting thing to note is that there is a very high variation in the number of times that a copy of an already fixed relocatable offset in dynamic data is used. This ranges from a few tens to tens of millions using these test cases. At the same time, we note that there does not seem to be any significant correlation of this number and the actual number of the reconstructed relocatable offsets.

5.4.2 Performance Overhead

Next, we focus on evaluating the performance overhead. As already mentioned, the only case where we expect our technique to affect the performance of a target application is during the first (or, few first) times we execute it, where most of the relocations are being discovered. Any consecutive execution should have a minimal runtime overhead impact.

Figure 5.3 shows the normalized slowdown for the first execution of the SPEC programs under our prototype (Discovery run) and another execution after the relocations have been discovered (Second run). In both cases, the slowdown is compared to a normal execution

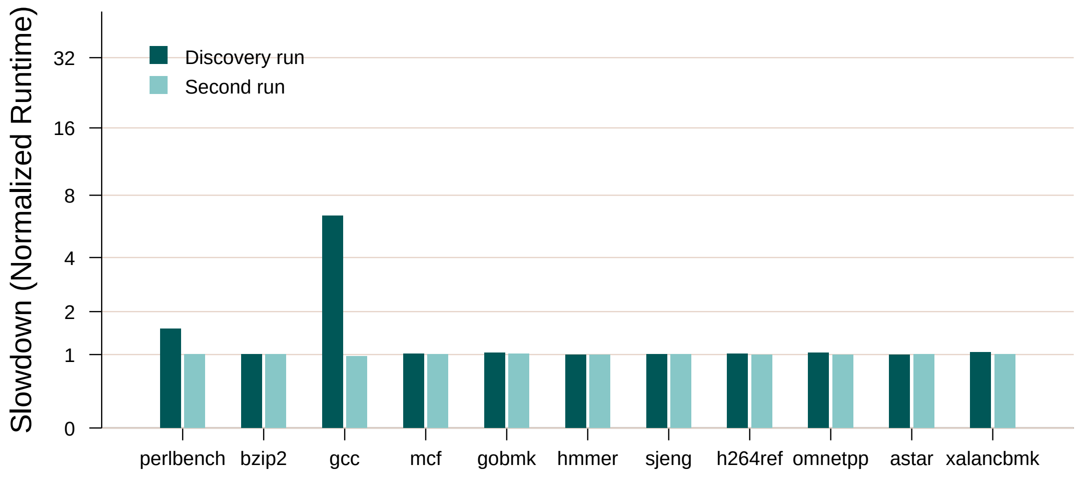


Figure 5.3: Normalized slowdown compared to normal execution (no relocation). Dark-colored bars show the slowdown during the first run, where most of the relocations are discovered and there are still copies of them in dynamic data. Light-colored bars show the slowdown during the second run (and any subsequent runs) where most of the relocations have already been discovered.

without relocating the program (baseline). Also, the input data used for this experiment was the reference dataset (i.e., the largest dataset), where the average completion time for each test program is a couple of minutes. As expected, we see that the overhead of the second run is minimal (less than 5% on average) and mostly attributed to the unoptimized way of applying the discovered relocation information. Currently, in our prototype implementation we relocate every offset separately. For each of them, we read its value, change the memory permissions, update its value and restore the memory permissions. The unusually high performance overhead that we observed when executing `gcc` is due to the fact that it contains a high number of relocatable offset copies in dynamic data (see Table 5.1). Although, that overhead does disappear in any consecutive execution, there is not much we can do at this point, except asking the user to restart the execution of the program in order to take advantage of the already discovered relocatable offsets. An alternative strategy is to ask the user to start with a very small input and progressively increase the workload of the program during the first few executions, until the majority of the relocations are discovered.

To demonstrate the effectiveness of that strategy, we applied it on the SPEC CPU2006

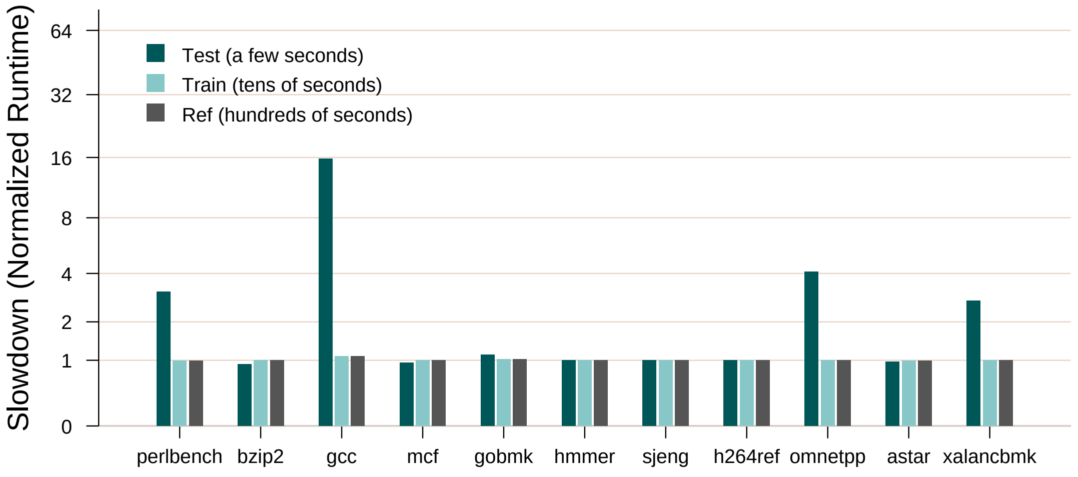


Figure 5.4: Avoiding the performance hit during the dynamic relocation discovery phase (first run) by gradually increasing the input size on each execution. The overall time in this case is much less compared to running a program using large input the first time.

benchmarks. These test programs come with three different inputs: a very small test dataset used for verifying the functionality of the programs, a medium-sized train set used for feedback-directed optimizations and the reference dataset, which is much larger that the other two. For all the results up to this point, we have used the reference dataset. Figure 5.4 shows the normalized slowdown of applying our technique to the same SPEC programs, but while increasing the workload (from test, to train and reference) this time. Also, during each execution, we allow our prototype to use any reconstructed relocation information that has been discovered from previous executions. The slowdown of the reference dataset is much less compared to the one reported in Figure 5.3. Moreover, the overall discovery phase (which is now broken down to three executions) is much quicker compared to Figure 5.3, in absolute numbers. Even though `gcc` seems to have a larger slowdown with the test dataset than before, this accounts for 22 seconds, plus a few minutes for the next two executions, compared to 48 minutes when using the large reference dataset during the first execution.

5.4.3 Use Cases

The final part of our evaluation focuses on the feasibility of applying our technique on popular, real-world applications. For this purpose, we installed older versions of both

Internet Explorer and Adobe Reader, where the relocation info of their EXE files was stripped. The exact versions we used were 6.0.2900.5512 and 8.1.2, respectively. In both cases, the code size of the non-relocatable EXE was relative small, approximately 10KB. Using our prototype implementation of our technique we were able to successfully relocate the code segments to a new and random location, while not breaking the functionality of the applications. The number of relocatable offsets for which we reconstructed their relocation information was 18 for Internet Explorer and 3 for Adobe Reader. Although it is just a small number of relocations, reconstructing this information is crucial in protecting these applications.

Chapter 6

Conclusions

6.1 Closing Remarks

The increasing number of exploits against Windows applications that rely on return-oriented programming to bypass exploit mitigations such as DEP and ASLR, necessitates the deployment of additional protection mechanisms that can harden imminently vulnerable third-party applications against these threats. Exploit mitigation add-ons that can be readily enabled for the protection of already installed applications are among the most practical ways for deploying additional layers of defenses on existing systems.

Towards this goal, we have presented in-place code randomization, a technique that offers probabilistic protection against ROP attacks, by randomizing the code of third-party applications using various narrow-scope code transformations. Our approach is practical: it can be applied directly on third-party executables without relying on debugging information, and does not introduce any runtime overhead. At the same time, it is effective: our experimental evaluation using in-the-wild ROP exploits and two automated ROP code construction toolkits shows that in-place code randomization can thwart ROP attacks against widely used applications, including Adobe Reader on Windows 7, and can prevent the automated generation of ROP code resistant to randomization. Our prototype implementation is publicly available, and as part of our future work, we plan to improve its randomization coverage using more advanced data flow analysis methods, and extend it to support ELF and 64-bit executables.

To be even more usable in practice, any such solution could benefit from being completely transparent and not impacting in any way the normal operation of the protected applications. Starting on this basis, we have presented the design and implementation of kBouncer, a transparent ROP exploit mitigation based on the identification of distinctive attributes of return-oriented or jump-oriented code that are inherently exhibited during execution. Built on top of the Last Branch Recording (LBR) feature of recent processors for tracking the execution of indirect branches at critical points during the lifetime of a process, kBouncer introduces negligible runtime overhead, and does not require any modifications to the protected applications. We believe that the most important advantage of the proposed approach is its practical applicability. We demonstrate that our prototype implementation for Windows 7 can effectively protect complex, widely used applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader, against in-the-wild ROP exploits, without any false positives.

Finally, we recognized the importance of relocation information in both enabling ASLR, which has proven to be a very effective mitigation against code reuse attacks and in improving disassembly coverage, which in tern improves the accuracy of many mitigations that depend on it. As a step towards addressing this limitation in programs where relocation information is stripped, we designed and implemented a technique to dynamically reconstruct this missing information, which effectively enables ASLR even on programs that are otherwise incompatible. The results of our experimental evaluation focusing on performance measurements and use cases with real-world applications clearly show the practicality of the proposed approach.

6.2 Future Directions

Although our work so far demonstrates the practicality of our defense techniques, there is still room for improvement. In this final section we outline some of the most important directions that require further research.

The main idea behind in-place code randomization is that an attacker will most probably avoid re-using any part of the code that might have been transformed. In case a ROP exploit does depend on transformed parts of code (e.g., if the attacker was not aware of it), in-place code randomization will prevent it by crashing the application. That is because if an attacker tries to re-use a piece of code that has been randomized, this will probably lead to executing an illegal instruction or accessing out of bounds memory, etc. Being able to detect ROP exploit attempts is a valuable feature, because, currently, there is no way to distinguish whether the source of an application crash is due to a bug or due to a failed ROP exploit attempt. This will not only make the technique more user-friendly, but informing users about an attempted attack allows them to take further actions — like quarantining the input file, blocking its sender, and so on. Ideally, an attack attempt could be detected at runtime by inserting extra analysis code. However, this is not possible as no new instructions are added by in-place code randomization. A more proper solution that is aligned with the technique's goals is to perform the analysis post-mortem: given a crash dump report, try to identify the instruction which caused the application to crash and then check if it was transformed or not.

The most important effectiveness metric of in-place code randomization is coverage. Although achieving full coverage is a very challenging task, there is still space for improvement by adding more transformation schemes. An area that current transformation do not affect as much is basic blocks of a relatively small size. The intuition is that basic blocks with a few instructions are usually left out, because their instructions have serial dependencies, or there are cases where a single register is preserved and so on. A new transformation scheme that would greatly improve coverage in these cases would be to entirely relocate small basic blocks to a new and random locations. Although this may not seem to strictly follow the in-place characteristic, the important part is for the basic block boundaries of the original code to remain the same. Thus, from this point of view, it is still in-place. At a high level, to implement this scheme we would need to copy the instructions of a basic block to a random location, replace its first instruction in the original location with a transfer control and fill up the rest of it with `nop` instructions — alternatively, a more optimized implementation could accommodate smaller basic blocks in the empty spaces. Another important factor that clearly affects randomization coverage is the instruction set itself. It would be very interesting to evaluate the coverage for the same transformations on other architectures

(ARM, x86-64, etc.), when applicable, or even design new transformations specific to these new architectures.

Hardware features offer many great advantages when used for ROP protection as we showed through this work. However, as discussed in Section 4.5, a determined attacker might still be able to get around kBouncer. Our current design incorporates two indirect branch checks, for illegal returns and chains of gadgets, but it can be easily extended to include more. As an example, kBouncer could greatly reduce the number of allowed call-preceded gadgets by refining the illegal return check. Instead of only checking for a call instruction before the return target, we could also verify the call's target, if it is a direct one. Additionally, to prevent an attacker from using a long-enough useful gadget, we could slightly change the gadget definition in the chain check. Instead of having a maximum instructions length, we could score gadgets based on their usefulness. A gadget that clobbers 6 registers is probably less useful than a gadget that does not clobber any, irrespectively of their length. Both extension of course require an in-depth study, but still demonstrate that there is greater potential in kBouncer.

Despite having the numerous advantages that were previously described in Section 4.1.1, LBR feature's main limitation is its size. The first generation of processors that included this feature had a stack of only 4 registers. The size has since gradually expanded to 16, but still it is limited. The disadvantage of having a small LBR stack is that it theoretically allows the attackers to find a long enough execution path leading to a system call that overwrites all the LBR entries with legitimate branches. Although finding such a path is significantly difficult, as the arguments to the system call have to be preserved, it still poses a threat. Thus, the more entries the LBR can hold, the longer the execution path has to be and the longer it is the more difficult to find. To overcome this limitation we could push extra check points further back in the execution paths leading to system calls, effectively extending the size of the LBR stack. The idea is, given an LBR instance with the last 16 executed branches, we examine the source of the oldest branch and replace it with a check point by patching the code. Next time the same execution path is followed, we will process the LBR contents at this new check point and access the 16 previous branches. For this case, this is equal to having an LBR stack of size 32. Of course, there are a few limitations and design choices that need to be made, but the overall idea seems quite promising.

Finally, we should not forget that the original purpose of the LBR feature was to assist debugging and performance issues. A question that naturally rises here is: *How would one design a similar processor feature, specifically for building protection policies?* A fundamental limiting factor, apart from the size of the LBR, is its lack of flexibility. On the other hand, providing support for richer policies at the hardware level than simply recording branches might come with non-negligible performance overhead. This is just one of the trade offs that need to be considered. We do recognize that designing such a mechanism is not trivial. However, even supporting a simple policy language would be better compared to today's standards where one has to wait a few years until a new security policy is implemented in the next generation of a processor (e.g., Intel's Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP) [Intel, 2014]).

Bibliography

[Abadi *et al.*, 2005] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.

[Aho *et al.*, 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Bania, 2005] Piotr Bania. Windows Syscall Shellcode, 2005. `http://www.securityfocus.com/infocus/1844`.

[Barrantes *et al.*, 2003] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.

[Baumgartner, 2010] Kurt Baumgartner. The ROP pack. In *Proceedings of the 20th Virus Bulletin International Conference (VB)*, 2010.

[Bennett *et al.*, 2013] James Bennett, Yichong Lin, and Thoufique Haq. The Number of the Beast, 2013. `http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html`.

[Bhatkar *et al.*, 2003] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[Bhatkar *et al.*, 2005] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[Bletsch *et al.*, 2011a] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[Bletsch *et al.*, 2011b] Tyler Bletsch, Xuxian Jiang, Vince Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[Bouchez, 2009] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École normale supérieure de Lyon, April 2009.

[Bremer, 2012] Jurriaan Bremer. Intercepting System Calls on x86_64 Windows, 2012. `http://jbremer.org/intercepting-system-calls-on-x86_64-windows/`.

[Buchanan *et al.*, 2008] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[Carlini and Wagner, 2014] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[Checkoway *et al.*, 2009] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage. In *Proceedings of the 2009 conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE)*, 2009.

[Checkoway *et al.*, 2010] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[Chen *et al.*, 2005] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[Chen *et al.*, 2009] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[Cheng *et al.*, 2014] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[Cohen, 1993] Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12:565–584, October 1993.

[Corelan Team, a] Corelan Team. Corelan ROPdb. `https://www.corelan.be/index.php/security/corelan-ropdb/`.

[Corelan Team, b] Corelan Team. Mona. `http://redmine.corelan.be/projects/mona`.

[Davi *et al.*, 2009] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.

[Davi *et al.*, 2011] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A practical protection tool to protect against return-oriented programming. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[Davi *et al.*, 2013] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIACCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[Davi *et al.*, 2014] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[Designer, 1997] Solar Designer. Getting around non-executable stack (and fix), 1997. `http://seclists.org/bugtraq/1997/Aug/63`.

[Dullien *et al.*, 2010] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[El-Khalil and Keromytis, 2004] Rakan El-Khalil and Angelos D. Keromytis. Hydan: Hiding information in program binaries. In *Proceedings of the International Conference on Information and Communications Security, (ICICS)*, 2004.

[Erlingsson, 2007] Úlfar Erlingsson. Low-level software security: Attack and defenses. Technical Report MSR-TR-07-153, Microsoft Research, 2007. `http://research.microsoft.com/pubs/64363/tr-2007-153.pdf`.

[Field,] Scott Field. An introduction to kernel patch protection. `http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx`.

[Fog,] Agner Fog. Calling conventions for different C++ compilers and operating systems. `http://agner.org/optimize/calling_conventions.pdf`.

[Forrest *et al.*, 1997] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[Fratric, 2012] Ivan Fratric. Runtime prevention of return-oriented programming attacks, 2012. `https://code.google.com/p/ropguard/`.

[Fresi Roglia *et al.*, 2009] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[Göktas *et al.*, 2014a] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35rd IEEE Symposium on Security & Privacy (S&P)*, 2014.

[Göktaş *et al.*, 2014b] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[Google, 2011] Google. Syzygy - profile guided, post-link executable reordering, 2011. `http://code.google.com/p/sawbuck/wiki/SyzygyDesign`.

[Guilfanov, 2008a] Ilfak Guilfanov. Decompilers and beyond. Black Hat USA, 2008.

[Guilfanov, 2008b] Ilfak Guilfanov. Jump tables, 2008. `http://www.hexblog.com/?p=68`.

[Harris and Miller, 2005] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33:63–68, December 2005.

[Hex-Rays,] Hex-Rays. IDA Pro Disassembler. `http://www.hex-rays.com/idapro/`.

[Hiser *et al.*, 2012] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.

[Hu *et al.*, 2009] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, 2009.

[Hund *et al.*, 2009] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[Hund *et al.*, 2013] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[Hunt and Brubacher, 1999] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[Immunity,] Immunity. Immunity Debugger. `http://www.immunityinc.com/products-immdbg.shtml`.

[Immunity, 2010] Immunity. White Phosphorus Exploit Pack, 2010. `https://www.immunityinc.com/products-whitephosphorus.shtml`.

[Intel, 2014] Intel. Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2014.

[Johnson, 2011] Richard Johnson. A castle made of sand: Adobe Reader X sandbox. CanSecWest, 2011.

[Jurczyk, 2011] Mateusz Jurczyk. Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8), 2011. `http://j00ru.vexillium.org/ntapi/`.

[Kayaalp *et al.*, 2012] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 94 –105, 2012.

[Kc *et al.*, 2003] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.

[Kil *et al.*, 2006] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[Krahmer, 2005] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[Kruegel *et al.*, 2004] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[Li *et al.*, 2010] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.

[Li, 2011] Haifei Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.

[Luk *et al.*, 2005] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[Metasploit,] Metasploit. Penetration testing software. `http://www.metasploit.com`.

[Metasploit, 2010] Metasploit. Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow, 2010. `http://www.metasploit.com/modules/exploit/windows/fileformat/adobe_cooltype_sing`.

[Metasploit, 2012a] Metasploit. Adobe Flash Player 11.3 Kern Table Parsing Integer Overflow, 2012. `http://www.metasploit.com/modules/exploit/windows/browser/adobe_flash_otf_font`.

[Metasploit, 2012b] Metasploit. MS12-063 Microsoft Internet Explorer execCommand Use-After-Free Vulnerability, 2012. `http://www.metasploit.com/modules/exploit/windows/browser/ie_execcommand_uaf`.

[Microsoft, a] Microsoft. Enhanced Mitigation Experience Toolkit. `http://www.microsoft.com/emet`.

[Microsoft, b] Microsoft. /ORDER (put functions in order). `http://msdn.microsoft.com/en-us/library/00kh39zz.aspx`.

[Microsoft, c] Microsoft. Profile-guided optimizations. `http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx`.

[Microsoft, d] Microsoft. Windows api list. `http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx`.

[Microsoft, e] Microsoft. Windows Debugging API. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms679303(v=vs.85).aspx`.

[Microsoft, f] Microsoft. Windows filtering platform. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx`.

[Microsoft, g] Microsoft. Windows XP SP3 and Office 2003 Support Ends April 8th, 2014. `http://www.microsoft.com/en-us/windows/enterprise/endofsupport.aspx`.

[Microsoft, 2012] Microsoft. Microsoft BlueHat prize contest official rules. `http://www.microsoft.com/security/bluehatprize/rules.aspx`, April 2012.

[Miller *et al.*, 2011] Matt Miller, Tim Burrell, and Michael Howard. Mitigating software vulnerabilities, July 2011. `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788`.

[Muchnick, 1997] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Nanda *et al.*, 2006] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[Nate_M, 2011] Nate_M. Mplayer (r33064 lite) buffer overflow + rop exploit, 2011. `http://www.exploit-db.com/exploits/17124/`.

[Nepenthes, 2007] Nepenthes. Common Shellcode Naming Initiative, 2007. `http://nepenthes.carnivore.it/csni`.

[Nergal, 2001] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.

[Newsham, 2000] Tim Newsham. Non-exec stack, 2000. `http://seclists.org/bugtraq/2000/May/90`.

[Node, 2010] Node. Integard Pro 2.2.0.9026 (Win7 ROP-Code Metasploit Module), 2010. `http://www.exploit-db.com/exploits/15016/`.

[Onarlioglu *et al.*, 2010] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[Pappas *et al.*, 2012] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.

[Pappas *et al.*, 2013] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[Pappas *et al.*, 2014] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Dynamic reconstruction of relocation information for stripped binaries. In *Proceedings* *of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[Parkour, 2011] Mila Parkour. An overview of exploit packs (update 9) April 5 2011. 2011. `http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs- update.html`.

[Parvez, 2013] Parvez. MS13-008 Microsoft Internet Explorer CButton Use-After-Free Vulnerability, 2013. `http://www.greyhathacker.net/?p=641`.

[PaX Team, 2001] PaX Team. Address space layout randomization, 2001. `http://pax.grsecurity.net/docs/aslr.txt`.

[Pewny and Holz, 2013] Jannik Pewny and Thorsten Holz. Control-flow restrictor: compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 309–318, 2013.

[Pietrek, 2002] Matt Pietrek. An in-depth look into the Win32 portable executable file format, part 2, 2002. `http://msdn.microsoft.com/en-us/magazine/cc301808.aspx`.

[Polychronakis *et al.*, 2009] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[Portnoy, 2013] Aaron Portnoy. Bypassing all of the things. SummerCon, 2013.

[Rescorla, 2003] Eric Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, August 2003.

[Rosenberg, 2011] Dan Rosenberg. Defeating Windows 8 ROP Mitigation, 2011. `http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/`.

[Russinovich, 2006] Mark Russinovich. Inside native applications, November 2006. `http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx`.

[Saxena *et al.*, 2008] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization (CGO)*, 2008.

[Schuster *et al.*, 2014] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maa, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[Schwartz *et al.*, 2011] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[Serna, 2012] Fermin J. Serna. CVE-2012-0769, the case of the perfect info leak, February 2012. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

[Shacham *et al.*, 2004] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.

[Shacham, 2007] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007.

[Skaletsky *et al.*, 2010] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems*, 2010.

[Skape and Skywing, 2005] Skape and Skywing. Bypassing Windows hardware-enforced DEP. *Uninformed*, 2, September 2005.

[Skape, 2003] Skape. Understanding windows shellcode, 2003. `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`.

[Skape, 2007] Skape. Locreate: An anagram for relocate. *Uninformed*, 6, 2007.

[Smithson *et al.*, 2010] Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. Binary rewriting without relocation information. Technical report, University of Maryland, 2010. `http://www.ece.umd.edu/~barua/without- relocation-technical-report10.pdf`.

[Snow *et al.*, 2013] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[Soffa *et al.*, 2011] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[Solé, 2008] Pablo Solé. Defeating DEP, the Immunitiy Debugger way, 2008. `http://www.immunitysec.com/downloads/DEPLIB.pdf`.

[Solé, 2010] Pablo Solé. Hanging on a ROPe, 2010. `http://www.immunitysec.com/downloads/DEPLIB20_ekoparty.pdf`.

[SPEC, 2006] SPEC. SPEC CPU2006 Benchmark. `http://www.spec.org/cpu2006`, 2006.

[Summers, 2014] Nick Summers. ATMs Face Deadline to Upgrade From Windows XP. `http://www.businessweek.com/articles/2014-01-16/atms-face-deadline-to-upgrade-from-windows-xp`, 2014.

[Ször, 2005] Péter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[UKX, 2014] UK government pays Microsoft £5.5m to extend Windows XP support. `http://www.theguardian.com/technology/2014/apr/07/uk-government-microsoft-windows-xp-public-sector`, 2014.

[Varol and Rotem, 1981] Yaakov L. Varol and Doron Rotem. An algorithm to generate all topological sorting arrangements. *Comput. J.*, 24(1):83–84, 1981.

[Vasudevan *et al.*, 2011] A. Vasudevan, Ning Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS)*, 2011.

[Vreugdenhil, 2010] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. 2010. `http://vreugdenhilresearch.nl/Pwn2Own12010-Windows7-InternetExplorer8.pdf`.

[Wartell *et al.*, 2012] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, October 2012.

[Wicherski, 2013] Georg Wicherski. Taming ROP on Sandy Bridge. SyScan, 2013.

[Wine,] Wine. Winehq: Run windows applications on linux, bsd, solaris and mac os x. `http://www.winehq.org`.

[Wurster *et al.*, 2005] Glenn Wurster, P.C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, volume 0, pages 127–138, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[Xia *et al.*, 2012] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[Yuan *et al.*, 2011] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*, 2011.

[Zhang and Sekar, 2013] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium*, pages 337–352, Berkeley, CA, 2013. USENIX.

[Zhang *et al.*, 2013] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[Zovi, 2010a] Dino A. Dai Zovi. Mac OS X return-oriented exploitation. RECON, 2010.

[Zovi, 2010b] Dino A. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010.