# Quantitative Metrics and Measurement Methodologies for System Security Assurance

Salman Ahmed

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Danfeng (Daphne) Yao, Chair
Matthew Hicks
Gang Wang
Patrick R. Schaumont
Fabian Monrose

December 9, 2021
Blacksburg, Virginia

Keywords: Security Measurement, Attack Surface Quantification, Metrics, Methodologies, Attack Vectors, Data Pointers, Taint Analysis, LLVM, ROP, JITROP, Data-Oriented Attacks, Gadgets, ASLR, Pointer Authentication, Memory Tagging

# Quantitative Metrics and Measurement Methodologies for System Security Assurance

Salman Ahmed

(ABSTRACT)

Proactive approaches for preventing attacks through security measurements are crucial for preventing sophisticated attacks. However, proactive measures must employ qualitative security metrics and systemic measurement methodologies to assess security guarantees, as some metrics (e.g., entropy) used for evaluating security guarantees may not capture the capabilities of advanced attackers. Also, many proactive measures (e.g., data pointer protection or data flow integrity) suffer performance bottlenecks. This dissertation identifies and represents attack vectors as metrics using the knowledge from advanced exploits and demonstrates the effectiveness of the metrics by quantifying attack surface and enabling ways to tune performance vs. security of existing defenses by identifying and prioritizing key attack vectors for protection. We measure attack surface by quantifying the impact of fine-grained Address Space Layout Randomization (ASLR) on code reuse attacks under the Just-In-Time Return-Oriented Programming (JITROP) threat model. We conduct a comprehensive measurement study with five fine-grained ASLR tools, 20 applications including six browsers, one browser engine, and 25 dynamic libraries. Experiments show that attackers only need several seconds (1.5-3.5) to find various code reuse gadgets such as the Turing Complete gadget set. Experiments also suggest that some code pointer leaks allow attackers to find gadgets more quickly than others. Besides, the instruction-level single-round randomization can restrict Turing Complete operations by preventing up to 90% of gadgets. This dissertation also identifies and prioritizes critical data pointers for protection to enable the capability to tune between performance vs. security. We apply seven rule-based heuristics to prioritize externally manipulatable sensitive data objects/pointers. Our evaluations using 33 ground truths vulnerable data objects/pointers show the successful detection of 32 ground truths with a 42% performance overhead reduction compared to AddressSanitizer. Our results also suggest that sensitive data objects are as low as 3%, and on average, 82% of data objects do not need protection for real-world applications.

# Quantitative Metrics and Measurement Methodologies for System Security Assurance

Salman Ahmed

(GENERAL AUDIENCE ABSTRACT)

Proactive approaches for preventing attacks through security measurements are crucial to prevent advanced attacks because reactive measures can become challenging, especially when attackers enter sophisticated attack phases. A key challenge for the proactive measures is the identification of representative metrics and measurement methodologies to assess security guarantees, as some metrics used for evaluating security guarantees may not capture the capabilities of advanced attackers. Also, many proactive measures suffer performance bottlenecks. This dissertation identifies and represents attack elements as metrics using the knowledge from advanced exploits and demonstrates the effectiveness of the metrics by quantifying attack surface and enabling the capability to tune performance vs. security of existing defenses by identifying and prioritizing key attack elements. We measure the attack surface of various software applications by quantifying the available attack elements of code reuse attacks in the presence of fine-grained Address Space Layout Randomization (ASLR), a defense in modern operating systems. ASLR makes code reuse attacks difficult by making the attack components unavailable. We perform a comprehensive measurement study with five fine-grained ASLR tools, real-world applications, and libraries under an influential code reuse attack model. Experiments show that attackers only need several seconds (1.5-3.5) to find various code reuse elements. Results also show the influence of one attack element over another and one defense strategy over another strategy. This dissertation also applies seven rule-based heuristics to prioritize externally manipulatable sensitive data objects/pointers – a type of attack element – to enable the capability to tune between performance vs. security. Our evaluations using 33 ground truths vulnerable data objects/pointers show the successful identification of 32 ground truths with a 42% performance overhead reduction compared to AddressSanitizer, a memory error detector. Our results also suggest that sensitive data objects are as low as 3% of all objects, and on average, 82% of objects do not need protection for real-world applications.

# Dedication

*To my wife (Nasrin Sultana), who sacrificed most and inspired me all the time to get through each step of the most challenging journey of my life and gave me a precious gift during this journey, our son, Nazhan Ahmed.*

*To my parents (Akbar Ali and Jabeda Begum) for their all-time unyielding support, love, and encouragement.*

# Acknowledgments

I am forever grateful to my parents for their continuous support and encouragement for my education. Being in a society where sons tend to support their families right after the bachelor's degree, my parents never asked me for that support. They always encouraged me to move forward and pursue my passion, no matter how bad the situation was for them. This journey would not have been possible without their support and encouragement.

I want to express my sincere gratitude to my Ph.D. advisor Dr. Danfeng (Daphne) Yao, for believing me and guiding me throughout my Ph.D. study with her continuous support and extraordinary patience. Her prompt and timely suggestions have enabled me to complete this dissertation on time. Her support went beyond academic settings. She trained me not only how to do research but also how to make myself fit for the next chapter of my life from scratch. She taught me to express myself, be professional in valuing other people and their work, and prioritize my goals. My sincerest thanks to my committee members, who have encouraged me and helped pave the pathway. Thanks to Dr. Matthew Hicks, Dr. Gang Wang, Dr. Patrick R. Schaumont, and Dr. Fabian Monrose for serving in my Ph.D. committee and guiding me over the years. Their insights, feedback, and advice were influential and essential for shaping the dissertation.

I would also like to thank all my mentors, collaborators, and fellow labmates, including Dr. Gang Tan, Dr. Kevin Snow, Dr. Long Chen, Dr. Sazzadur Rahaman, Dr. Hans Liljestrand, Dr. Haipeng Cai, Dr. Trent Jaeger, Dr. N. Asokan, Dr. Michael Le, Dr. Dan Williams, Dr. Hani Jamjoom, Dr. Xiaodong Yu, Ya Xiao, Sharmin Afrose, Miles Frantz, Wenjia Song, Tanmoy Sarkar Pias, Jingyuan Qi, and Wyatt Sweat.

Finally, my special thanks to my wife, Nasrin Sultana, who encouraged, inspired, and pushed me towards my goals with her selfless sacrifices.

**Funding Acknowledgment:** The work in this dissertation was supported in part by the National Science Foundation under grant No. CNS-1838271, Grant No. CNS- 1929701, Grants OAC-1541105, and CNS-1801534, and by the Office of Naval Research under Grant ONR-N00014-17-1-2498.

**Declaration of Collaboration:** In addition to my advisor Danfeng (Daphne) Yao, the research presented in this dissertation benefited from several collaborators:

- Ya Xiao (VT), Gang Tan (PSU), Kevin Snow (ZeroPoint Dynamics), and Fabian Monrose (UNC) contributed to the work included in Chapter 3.
- Long Chen (ClemsonU), Hans Liljestrand (UW), Thomas Nyman (Aalto University), Haipeng Cai (WSU), Trent Jaeger (PSU), and N. Asokan (UW) contributed to the work in Chapter 4.
- Hans Liljestrand (UW) and N. Asokan (UW) contributed to the work included in Chapter 5.

# List of Publications From This Thesis

1. **[ACM TOPS'21]** Long Cheng, **Salman Ahmed**, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. ACM Trans. Priv. Secur. 24, 4, Article 26 (November 2021), 36 pages. DOI:https://doi.org/10.1145/3462699

2. **[ACM CCS'20] Salman Ahmed**, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20). Association for Computing Machinery, New York, NY, USA, 18031820. DOI:https://doi.org/10.1145/3372297.3417248

3. **[IEEE SecDev'19]** Long Cheng, Hans Liljestrand, **Salman Ahmed**, Thomas Nyman, Trent Jaeger, N. Asokan, and Danfeng Yao, "Exploitation Techniques and Defenses for Data-Oriented Attacks," 2019 IEEE Cybersecurity Development (SecDev), 2019, pp. 114-128, doi: 10.1109/SecDev.2019.00022.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem Definitions

Despite the protections such as stack canaries [50], Write XOR execute or W$\oplus$X (aka No-execute (NX) [204] or Data Execution Prevention (DEP) [65]), and Address Space Layout Randomization (ASLR) [199] deployed in modern computer systems, we have observed many advanced and influential code reuse attacks (CRAs). Most of these CRAs utilize Return-Oriented Programming (ROP) technique. These advanced and influential attacks include Just-In-Time ROP or JIT-ROP [185], Blind ROP or BROP [18], Address Oblivious Code Reuse or AOCR [163], CrashResistant Oriented Programming or CROP [81], Position Independent ROP or PIROP [89], Return to Zombie Gadget or ZombieG [186], Counterfeit Object-Oriented Programming or COOP [169], Typed ROP or TROP [75], Control-Flow Bending or CFB [27], StackDefiler [46], Back To The Epilogue or BATE [17], and others [28, 31]. Each of these attacks has unique capabilities. For example, JIT-ROP [185] enables one to reuse code even under fine-grained address space layout randomization (ASLR). JIT-ROP attacks can discover new code pages dynamically, by leveraging control-flow transfer instructions, such as *call* and *jmp*.

To protect systems from these advanced attacks, researchers have proposed variants or fine-grained version of existing defenses (*e.g.*, fine-grained ASLR [47, 60, 62, 84, 92, 95, 97, 98, 108, 111, 151, 157, 213], re-randomization [16, 36, 38, 125, 216], execute-only memory(XOM) (aka destructive read) [11, 53, 197, 215]) and Code Pointer Integrity (CPI) [12, 49, 115, 126, 130]. However, many of these defenses focus on specific attacks, e.g., the XOM-based defenses aim to hinder the JIT-

ROP's repeated memory disclosure capability. Thus, XOM-based defenses evaluate their security guarantees only based on stopping disclosure attacks on executable binaries without rigorous and metric-based evaluation. That is why we have observed a code inference attack [186] that can work in the presence of XOM-based defenses.

Besides, some metrics used for evaluating security guarantees of defenses do not capture the capabilities of advanced attackers. As a result, new attacks emerge to bypass those defenses. For example, both coarse-grained code randomizations (e.g., PaX ASLR [199]) and fine-grained code randomizations (e.g., Selfrando [47], Compiler-assisted Code Randomization [111], Remix [38], STIR [213], ILR [97] and ASLP [108]) use entropy to measure the effectiveness of hindering code-reuse attacks. However, such an entropy measure is not useful under the JIT-ROP threat model, as chunks of code are still available. Inclusion of distances between permuted functions or basic blocks for computing entropy would not work either, because the code's semantic connectivity (e.g., through *call* and *jmp*) is still not captured. Code connectivity is what JIT-ROP attacks leverage to discover code pages. Metrics and measurement methodologies that accurately reflect JIT-ROP capabilities are more meaningful under the JIT-ROP model. Thus, the fundamental needs are representative security metrics and methodologies for evaluating defenses. This is why recently we have observed a new metric called tunable entropy [157] to measure security guarantee.

Another criterion to build robust and new defense mechanisms is to analyze the assumptions and requirements of threats [39, 40]. This analysis can lead to developing robust and efficient countermeasures. For example, a requirement for many typical and advanced code reuse attacks [18, 169, 185] is memory disclosure or more specifically code pointer leak. To stop attackers from fulfilling this requirement, researchers have proposed effective CPI-related countermeasures [12, 49, 115, 126, 130]. The CPI-based countermeasures are effective because these countermeasures incur as low as less than 1% overhead [126] and keep an application safe by protecting the integrity of a code pointer. Hardware-based techniques such as Intel's CET [103], ARM Pointer Authentication (PA) [158], and MPX [104]) can further reduce the overhead to make the slowdown imperceptible. For example, ARM PA [158] costs on average less than 0.5% for code

pointer authentication [121]. Thus, both software and hardware-based code pointer protection countermeasures are practical.

Due to this practical code pointer protection countermeasures [12, 49, 103, 104, 115, 126, 130, 158] as well as the advances towards practical CFI [24, 83, 124, 130, 132, 152, 222, 224], we anticipate a shift towards the use of data object/pointer manipulation as the attack vector as the manipulation works in the presence of these countermeasures. In recent years, we have also observed data-oriented attacks (also known as non-control attacks) [35, 100, 101, 133, 168, 196, 219] that manipulate data objects/pointers to exploit a system/application. While the software-based code pointer protection countermeasures are practical, software-based data object/pointer protection can cost a significant amount of overhead, from 48-116% [29, 134, 135]. Thus, software-based data object/pointer protection, in general, is not practical due to a high runtime overhead. On the other hand, hardware-based solutions can reduce the overhead significantly. For example, ARM pointer authentication [158] and Intel's MPX [104] cost on average around 19.5% [121] and 50% [145] overhead, respectively, for protecting data pointers. The 19.5% overhead for a hardware-based technique is still critical for performance-critical applications. Due to a huge number of data objects/pointers in an application compared to code pointers, one source of the overhead for protecting data objects/pointers is protecting many data objects/pointers that do not need protection as those objects/pointers do not lead to vulnerability. One way to reduce the performance overhead is to figure out the sensitive (*i.e.*, vulnerable) data objects/pointers and prioritize them for protection.

## 1.2 Contribution

The goal of this dissertation is to demonstrate the feasibility of methodologies for quantitatively measuring the security properties of advanced system defenses. In particular, we quantify the impact of fine-grained ASLR through attack surface measurement and improve the performance of in-memory data integrity defenses by identifying, quantifying, and prioritizing key attack vectors

extracted using knowledge from advanced exploits. To be specific, this dissertation aims to tackle three problems: *i)* designing security metrics and measurement methodologies for evaluating defense mechanisms, *ii)* analyzing the assumptions and requirements of advanced threats (e.g., data-oriented exploits), and *ii)* designing a data object/pointer prioritization technique to improve the existing defenses by reducing overhead by prioritizing sensitive data objects/pointers and filtering out non-sensitive ones.

To address the first problem, we have proposed four security metrics and four measurement methodologies to quantitatively evaluate the impact of fine-grained ASLR or code randomization defenses on various applications (see details on chapter 3). Besides, we have addressed some in-depth questions regarding the impact of fine-grained ASLR on code reuse attacks that have not been addressed before. For example, what impact do fine-grained ASLR have on the Turing-complete expressiveness of JIT-ROP payloads? How do attack vectors (e.g., code pointer leaks) impact the code reuse attacks? How would one compute the re-randomization interval effectively to defeat JIT-ROP attacks? We designed a measurement mechanism that allows us to perform JIT-ROP's code page discovery in a scalable fashion. This mechanism enables us to compare results from many programs and libraries under multiple ASLR conditions (coarse-grained, fine-grained function level, fine-grained basic block level, fine-grained instruction level, and register level). Our evaluation involves up to 20 applications, including six browsers, one browser engine, and 25 dynamic libraries. Our key experimental findings and technical contributions are summarized as follows.

- A multi-step attack workflow that captures the common tasks and goals in ASLR bypasses.
- We define multiple new concepts, e.g., minimum footprint gadgets, extended footprint gadgets, and quality of gadgets, and describe methods for evaluating important properties of ROP gadgets, e.g., register corruption rate. We also summarize and experiment with common and specialized gadget types used in recent attacks. These contributions are useful beyond this specific ASLR study.

- We provide a methodology to compute the upper bound $\mathcal{T}$ for re-randomization intervals. If the re-randomization interval is less than $\mathcal{T}$, then a JIT-ROP attacker is unable to obtain various gadget sets such as the Turing complete gadget set, priority gadget set, MOV TC gadget set, and gadgets from real-world payloads (see the definitions of gadget sets in Section 3.2). We compute the upper bound $\mathcal{T}$ by measuring the minimum time for an attacker to find a specific gadget set, i.e., the shortest time to reach gadget convergence for the gadget set. The upper bound ranges from 1.5 to 3.5 seconds in our tested applications such as *nginx*, *proftpd*, *firefox*, etc.

- Our findings show that starting code pointers do not have any impact (i.e., zero standard deviations) on the reachability from one code page to another. Every code pointer leak is equally viable for revealing an address space layout, suggesting that attackers' discovered gadgets eventually converge to a gadget set no matter where the starting pointer is.

- Our findings also show that the starting code pointers have an impact on the speed of convergence. That means the time needed for a JIT-ROP attacker to discover a gadget set varies with the locations of starting code pointers. In our experiments, the time for obtaining the Turing-complete gadget set ranges from 2.2 to 5.8 seconds.

- We also present a general methodology for quantifying the number of JIT-ROP gadgets. Our results show that a single-round instruction-level randomization scheme can limit the availability of gadgets up to 90% and break the Turing-complete operations of JIT-ROP payloads. Also, fine-grained randomization slightly degrades the gadget quality, in terms of register-level corruption.

  A stack has a higher risk of revealing dynamic libraries than a heap or data segment because our experiments show that stacks contain 16 more libc pointers than heaps or data segments on average. This finding indicates the necessity of randomizing stack over heap or global variables.

We systemized data-oriented attacks [35, 100, 101, 133, 168, 196, 219] with their assumptions/requirements and attack capabilities to address the second problem. Our work provides a comprehensive description of the assumptions and requirements of data-oriented attacks and a comparison of existing defenses known to prevent data-oriented attacks. Our analysis in this work offers new directions to look at data-oriented attacks because these attacks do not tamper with the control flow of a victim program and are far more advantageous than return-oriented programming. The main contributions of this work are as follows.

- We systemized various data-oriented exploits by classifying their exploit techniques.
- We discussed various automatic data-oriented exploit generation tools [105, 154, 172] and compared the tools in terms of their flexibility and practicality.
- We discuss various representative data-oriented exploits on real-world applications and existing defense approaches to prevent those exploits in different stages.

To address the third problem, i.e., prioritizing sensitive or vulnerable data objects/pointers, we develop a prioritization framework (see details on chapter 5). The goal of the framework is to prioritize data pointers that are sensitive and may potentially lead to vulnerability. Our goal is to ensure security through data object/pointer protection through various security mechanisms (e.g., ARM PA [158], MPX [104], Softbound [134], etc.) while keeping the overhead low. The main contributions of this work are as follows.

- We proposed a prioritization framework that is both *i)* generic and *ii)* adaptable. The generic nature of the framework ensures that it does not rely on underlying operating systems, platforms, or programming languages. The adaptability feature makes the framework adaptable with minimum changes so that it can be used with other defenses such as ARM PA [158], MPX [104], Softbound [134], AddressSanitizer [175], and many more. One key use case of the framework is to tune the performance vs security in an application.

- We extracted seven vulnerability- and exploit-driven rule-based heuristics to prioritize data objects/pointers that are sensitive and could potentially lead to vulnerabilities.

- We implemented our framework using eight analysis passes (one for taint analysis and seven for data object/pointer identification using our rules), and one instrumentation pass on top of LLVM 12. We also implemented one instrumentation pass for instrumenting our prioritized pointers using ARM PA [158] and modified AddressSanitizer [175] tool to support instrumenting only the prioritized data objects/pointers in addition to AddressSanitizer's default behavior.

- To evaluate the prioritization framework, we constructed ground truths by manually analyzing vulnerable programs considering local/global data objects/pointers, inter-functional analysis, and corner cases. We constructed 33 ground truths data objects from 18 programs including real-world server applications and 10 test cases from Software Assurance Reference (SAR) dataset.

- We found that as low as only 3% of total data objects are needed to protect for real-world applications. We achieved a 42% performance overhead reduction compared to AddressSanitizer while protecting 100% of the prioritized data objects. We can reduce the instrumentation size by around 62% and the number of pointers in Load/Store IR instructions (for ARM PA) by 56% and 96%, respectively, without compromising security.

The structure of this report is as follows. Chapter 2 related work on defense mechanisms against advanced ROP-based code reuse attacks, control-flow integrity, data-oriented attacks, pointer authentication, and sensitive data protection approaches. Chapter 3 presents our work on quantifying attack surface of various applications and libraries and assessing the impact of fine-grained ASLR or code randomization on code reuse attacks under the JIT-ROP threat model using quantitative metrics and measurement methodologies. Chapter 4 systemizes the knowledge of various data-oriented attacks by analyzing their requirements and assumptions. Chapter 5 presents our

technique for a data pointer prioritization framework including motivation, design, and evaluation. And, finally Chapter 6 outlines the guidelines and practical considerations that we need to make for making our measurement results effective as well as the practical implication of the measurements.

# Chapter 2

# Literature Review

The war games between attackers and defenders have been prevalent for decades. Due to legacy C/C++ code, memory corruption vulnerabilities are still prevalent in complex software such as browsers (e.g., Internet Explorer, Mozilla Firefox, etc.) and servers (e.g., Nginx, Apache, ProFTPD, etc.). Modern computer systems deploy various defenses (e.g., stack canaries [50], **W**rite XOR e**X**ecute or W$\oplus$X (aka **N**o-e**X**ecute (NX) [204] or **D**ata **E**xecution **P**revention (DEP) [65]), and Address Space Layout Randomization (ASLR) [199] to defend the attacks that exploit the memory corruption vulnerabilities. These defenses prevent attacks such as stack smashing [147] (e.g., return address overwrite), exception handler overwrite [122], and heap vulnerabilities (e.g., heap unlinking).

Unfortunately, the deployment of these defenses (*i.e.*, stack canaries, W$\oplus$, NX, and ASLR) could not repress advanced attackers because some attackers have proved their capabilities to bypass modern defenses. For example, coarse-grained code reuse attacks [153] or fine-grained code reuse attacks such as Return-Oriented Programming (ROP) [18, 28, 31, 62, 75, 81, 86, 161, 163, 178, 185, 186] can bypass the NX defense. At the same time, researchers have proposed various defenses such as fine-grained ASLR, Continuous address space randomization, Code Pointer Integrity (CPI), and Control Flow Integrity (CFI). Thus, the research conducted in the system security area primarily has two themes: 1) demonstrating attacks and 2) discovering countermeasures.

## 2.1 Attackers' and Defenders' Games

Attackers have primarily demonstrated two types of attacks: *i)* code injection attacks and *ii)* code reuse attacks. Besides, attackers also demonstrated attacks against a defense solution (*i.e.*, ASLR or CFI) to disclose the limitation of the defense.

### 2.1.1 Code Injection Attacks and Defenses

Code injection attacks [122, 147] inject shellcode into a writable segment of memory (*e.g.*, stack or heap) of a program and redirect the control flow to the injected shellcodes, usually by overwriting a return address or function pointer. Attacks that overwrite a return address are called stack smashing attacks. Stack canaries [50, 72] protect a program from stack smashing attacks. Besides, countermeasures such as StackGuard [50], Stack Shield [208], ProPolice [71] RAD [44], TRUSS [184], IBMAC [78], StackGhost [79], and Binary-Rewriting [156] aimed to protect return address integrity through shadow stack-based, compiler-based, and instrumentation-based techniques. To enforce canaries, compilers place a canary value before the stored return address in a stack and add a canary verification routine to function epilogues. Whenever the verification routine finds a mismatch in the function of a program, the program terminates by redirecting the control flow to a developer-defined exception handler. If the developer-defined exception handler is not present, the Operating System's exception handler handles the exception. In a consequent fashion, attackers overwrote the exception handler to construct an alternative control-flow hijack or the Structured Exception Handler (SEH) attacks [122]. In response, Windows Operating System (OS) added an overwrite protection feature called Structured Exception Handling Overwrite Protection (SEHOP). However, the major response came as hardware support called no-execute or NX bit [204]. CPUs segregate code and data regions based on the NX bit, and ensure the execution space protection. DEP or W$\oplus$X [65] utilizes the NX bit for marking some modules (*i.e.*, heap, stack, etc.) as non-executable. As a result, the writable segments of a binary are no longer executable. Thus, injecting

shellcodes in a stack or heap no longer works. To circumvent this DEP or W⊕X security feature, attackers invented code reuse attacks (e.g., return-to-libc).

### 2.1.2 Code Reuse Attacks and Defenses

Code reuse attacks (CRAs) can circumvent the DEP or W⊕X security feature. The first form of CRAs utilizes the whole library functions to construct exploits. Since the GNU C library, commonly known as Glibc or libc, is one of the largest libraries known to have diverse functionalities required by an attacker, attackers used the libc library for their attacks. This type of attack is called return-to-libc [112, 139, 153, 217]. Hovav Shacham extended the idea of code reuse by introducing a fine-grained code reuse technique called gadget chain. A gadget chain is a sequence of gadgets that can express arbitrary program logic. A gadget is a set of short instructions ending with the ret instruction. This technique is called Return Oriented Programming (ROP) [22, 155, 161, 178] and the gadgets are called ROP gadgets. Soon attackers adopted this technique and conducted many ROP-based code reuse attacks [18, 28, 31, 62, 75, 81, 86, 163, 185, 186] and its variants (*i.e.*, Jump Oriented Programming (JOP) [20]). Following are the four flavors of gadget-oriented CRAs.

- **Return-Oriented Programming (ROP) attacks**. Hovav Shacham first demonstrated the capability of ROP gadgets to construct Turing-complete operations [178]. The Turing-complete operations include memory operations, assignments, arithmetic operations, logical operations, control flow, function calls, and system calls [161]. Due to the capability of ROP gadgets, we have observed some influential ROP gadget-based attacks such as Just-In-Time ROP or JIT-ROP [185], Blind ROP or BROP [18], Address Oblivious Code Reuse or AOCR [163], CrashResistant Oriented Programming or CROP [81], Position Independent ROP or PIROP [89], and others [28, 31].

- **Jump-Oriented Programming (JOP) attacks**. JOP-based attacks [20, 33, 109, 131, 164] avoid the reliance of attacks on the stack and `ret` instructions by introducing JOP gadgets (a

sequence of instructions followed by a `jmp` instruction) as the building blocks for exploits. However, stack plays an important role in ROP-based attacks by allowing attackers to chain the ROP gadgets. To achieve the same capability, JOP-based attacks require dispatcher gadgets, in addition to the JOP gadgets.

- **Call-Oriented Programming (COP) attacks**. Carlini *et al.* [28] introduced and demonstrated the COP-based attacks using COP gadgets. COP gadgets are similar to ROP gadgets but end with `call` instructions. Unlike JOP-based attacks, COP attacks do not require dispatcher gadgets because `call` instructions usually use memory-indirect locations instead of values from registers as the targets. By preparing the memory in advance, attackers can chain the COP gadgets. Another COP-based attack is Pure-Call Oriented Programming or PCOP [165].

- **Counterfeit Object-Oriented Programming (COOP) attack**. Schuster *et al.* [169] introduced the COOP attack by demonstrating that C++ virtual functions can be chained together like gadgets to achieve malicious program behavior.

**Address Space Layout Randomization (ASLR)**. Code reuse or gadget-based attacks require the locations of shared-library functions or gadgets in memory. The PaX team introduced Address Space Layout Randomization (ASLR) [199] to make the code-reuse attacks difficult. ASLR, also known as coarse-grained ASLR, randomizes the base addresses of code (*i.e.*, .text) and data (*i.e.*, stack, heap, etc.) segments. As a result, it becomes difficult for attackers to know the locations of shared-library functions or gadgets without knowing how the locations of the modules are randomized by ASLR. ASLR is the most efficient and widely deployed security feature in modern operating systems. For example, OpenBSD added ASLR support in 2003, Linux in 2005, Android from Android 4.0, DragonFly BSD in 2010, iOS in 2011, Windows in 2007, NetBSD in 2009, macOS in 2007, and Solaris in 2012. The Position Independent Executable (PIE) option allows the main executable to be run as position-independent code, i.e., PIE relocates the code and data segments of the main executable.

ASLR makes the code reuse attacks difficult, but ASLR is vulnerable to information leaks. Since the coarse-grained ASLR only randomizes the base addresses of various segments and modules of a program, the internal layout of the segments and modules remains unchanged. Thus, a single leak can essentially reveal all the contents of a module. For example, the leak of a single function from the libc library can reveal all the other functions in libc because the offset of a function from the base address of libc is the same regardless of the different randomized base addresses of libc. An adversary can launch a basic ROP attack [198] using gadgets given a leaked address from the code segment of interest. The adversary only needs to adjust the addresses of pre-computed gadgets w.r.t. the leaked address.

**Defenses against ROP attacks**. Due to the prevalence of ROP attacks, researchers have proposed a wide range of countermeasures using both static software hardening and runtime monitoring techniques to prevent ROP attacks. Table 2.1 shows the key solutions. These solutions restrict the abnormal control flow transfers through return address protection, removing unaligned control transfer instructions, removing return address, ensuring the integrity of stack using shadow stack [79, 208] and detecting gadget chains by counting `ret` instructions or measuring gadget chain length.

Table 2.1: Countermeasures against ROP attacks. CRAs → Code Reuse Attacks.

| **Tool** | **Technique** | **Protection Against** | **Performance Overhead** |
|---|---|---|---|
| ROPGuard [80] | Performs runtime checks for ensuring integrity of stack pointer, invocation of critical functions, return address, and call stack when any critical function gets called. | ROP | <5% |
| G-Free [146] | Eliminates unaligned free-branch instructions and protects aligned free-branch instructions using alignment sleds, return address encryption, frame cookies and code rewriting. | ALL CRAs | 3% |
| DROP [32] | Detects an unusually high frequency of ret instructions | ROP | 5.3x |
| ROPdefender [64] | Ensures integrity of stack using shadow stack[79, 208] | ROP | 2x |
| Return-less [118] | Removes return instructions by replacing return addresses to return indices. | ROP | 4.6% |
| DyIMA [63] | Detects an unusually high frequency of ret instructions | ROP | |
| ROPecker [43] | Detects ROP gadget chains by analyzing the past and future execution flows of a process and extracting history of taken branches from Last Branch Record (LBR) registers. | ALL CRAs | 2.6% |
| kBouncer [150] | Detects abnormal control flow transfers by analyzing executed indirect branches at critical points recorded by Last Branch Record (LBR) registers. | ALL CRAs | 4% |

Most of the defense solutions discussed above (Table 2.1) are applicable for only ROP-based attacks. ROPecker [43] and kBouncer [150] are applicable for all control-oriented attacks but attackers can circumvent them [28, 87] using manually crafted long gadgets. G-Free [146] is also applicable for all control-oriented attacks, but it requires source code which is often unavailable. Besides, some of the solutions have high performance overheads.

**Leakage resilient defenses**. One prime requirement of code reuse attacks is information leaks. That means an attacker must derandomize the address space layout for mounting code-reuse or ROP attacks. To do so, an attacker first exploits a memory corruption vulnerability to leak information [9, 82, 96, 190, 192, 196] about the memory layout of a program. Then the attacker constructs an exploit leveraging the knowledge from the information leak and mount the attack using the same or another memory vulnerability. Under coarse-grained ASLR, a single leak can essentially reveal all the memory layout of a module. To stop revealing everything from a single leak, researchers introduced various leakage-resilient defenses. The leakage-resilient defenses unlock only a small portion of the code region and seriously limit an attack's ability to obtain gadgets for code reuse purposes. The leakage-resilient defenses fall into the following five categories.

***1) Fine-grained ASLR or code randomization***. Fine-grained ASLR, aka fine-grained code randomization or code diversification, relocates all the segments of the main executable of a process, including shared libraries, heap, stack, and memory-mapped regions, and restructures the internal layouts of these segments. Thus, simply adjusting the addresses of pre-computed gadgets (as in the basic ROP) no longer works. The granularity of fine-grained randomization varies, e.g., at the level of functions [47, 84, 108], basic blocks [38, 111, 213], instructions [97], or machine registers [52, 98]. Table 2.2 shows the summary of fine-grained code randomization defenses. However, few advanced attacks demonstrated their capability to perform memory disclosure attacks at runtime. For example, JIT-ROP [185] can recursively traverse code pages in an application using call and jump links. Other techniques such as BROP [18] and CROP [81] attacks can read memory contents dynamically using the so-called stack reading and memory probing techniques, respectively.

Table 2.2: Leakage-resilient defenses in four categories

| Tools | Techniques | Performance Overhead |
|---|---|---|
| **Fine-grained code randomization** | | |
| SBR [157] | - Limits the utility of any disclosed code address by partitioning a function<br>- Uses a configurable parameter to indicate partitions and offers tunable entropy | 2.26% [157] |
| Selfrando [47] | - Reorders functions at load time using metadata extracted by selfrando library | <1% [47] |
| CCR [111] | - Reorders functions and basic-block based on extracted metadata during compilation<br>- Keeps the same layout unless re-randomized again | 0.28% [111] |
| Zipr [95] | - Reorders the location of each instruction in an executable.<br>- Applies block-level instruction layout randomization during binary rewriting | <5% [95] |
| Multicompiler [98] | - Reorders functions and machine registers during link time optimization | 1% [98] |
| Isomeron [62] | - Performs code randomization with execution path randomization. | 19% [62] |
| ASLP [108] | - Permutes all sections, modules, and memory mapped regions by binary rewriting<br>- Reorders functions, data objects within their segments using relocation information | <1% [108] |
| ILR [97] | - Reorders the location of each instruction in an executable or library.<br>- A fall-through map guides the execution of instructions in the right order. | 13% [97] |
| ORP [151] | - Reorders instructions within a basic block.<br>- Employs narrow-scope code transformations using in-place code randomization | ∼0% [151] |
| Marlin [92] | - Extracts function symbols and shuffles the symbols | 0% [92] |
| XIFER [60] | - Transforms a control flow graph by randomizing contiguous basic blocks (bbls)<br>- Also, splits some bbls and injects dummy instructions in some bbls. | 1.2% [60] |
| STIR [213] | - Transforms a binary into a self-randomizable form using binary rewriting<br>- Self-randomizable form triggers randomization of basic blocks at program start | 1.6% [213] |
| ASR [84] | - Employs link-time transformation to randomize code and data for OSes<br>- Performs live re-randomization of a process' layout by runtime state migration. | <5% [84] |
| **Continuous randomization** | | |
| TASR [16] | - Randomizes a process' memory layout when the process inputs or outputs something | 2.1% [16] |
| Shuffler [216] | - Loads itself as a user space program and shuffles a process' functions continuously. | 14.9% [216] |
| Remix [38] | - Reorders the basic blocks within a function to avoid adjusting function pointers | 2.8% [38] |
| CodeArmor [36] | - Maps a code pointer to any one of multiple diversified code spaces<br>- Periodically changes the mapping from pointers to code space at runtime | 6.9% [36] |
| RuntimeASLR [125] | - Re-randomizes the address space of a child process after fork() with parent state | 0% [125] |
| STABILIZER [55] | - Re-randomizes functions, stack frames, and heap objects periodically at runtime. | <7% [55] |
| **Memory protection** | | |
| XnR [11] | - Executes code but restricts reading code as data by modifying page fault handlers<br>- Identifies bad code read through a CPU performance feature called demand paging | 2.2-3.4% [11] |
| NEAR [215] | - Allows the reading code, but does not allow to execute the read code<br>- A data swap process replaces any read instructions with invalid opcode<br>- Also, the process recovers the original opcodes on legitimate read and execution. | 4.7% [215] |
| Readactor [53] | - Enforces execute-only (XO) memory using Intel's Extended Page Tables (EPT)<br>- Converts code pointers into direct branches and hides them in an XO trampoline | 6.4% [53] |
| Heisenbyte [197] | - Marks code as execute-only and prevents reading after the execution of the code<br>- Allows legitimate reading of code by keeping a separate view of each page. | 16.5-18.3% [197] |
| **Code Pointer Integrity** | | |
| CPI [115] | - Identifies memory objects used to access code pointers and securely stores them | 2.9-8.4% [115] |
| Oxymoron [12] | - Replaces all the direct references to other code and data into indices.<br>- Redirects calls and jumps through a translation table that store actual mappings | 2.7% [12] |
| ASLR-Guard [126] | - Protects the leakage of code pointers by separating sensitive code or data pointers | <1% [126] |
| CCFI [130] | - Maintains the integrity of a CFG by ensuring the integrity of control-flow pointers<br>- Enforces integrity of control-flow pointers using cryptographic MACs for pointers. | 3-18% [130] |
| PointGuard [49] | - Ensures the integrity of pointers<br>- Encrypts when a pointer is initialized or modified and decrypts before use. | 0-20% [49] |

***2) Continuous code randomization***. Re-randomization (aka continuous code randomization) techniques [16, 36, 38, 55, 125, 216] continuously shuffle the address space at runtime. This continuous

shuffling breaks direct and indirect memory disclosure. It also breaks the runtime code discovery process by making the already discovered code pages obsolete. Re-randomization techniques are similar to fine-grained randomization techniques. However, re-randomization techniques pose one key challenge, which is the identification of all the pointer references and updating them at runtime. Table 2.2 shows a summary of continuous code randomization defenses.

***3) Memory protection***. The memory protection-related leakage-resilient defenses [11, 53, 197, 215] are variant of the W⊕X defense. The key goal of the memory protection leakage-resilient defenses is to prevent direct or indirect memory disclosure attacks [185]. The key goal of these memory protection techniques is to ensure execute-only memory for code pages. Table 2.2 shows a summary of memory protection-related leakage-resilient defenses.

However, attackers can still access zombie gadgets [186] that are available after applying destructive read defenses (i.e., XnR [11], NEAR [215], Readactor [52], and Heisenbyte [197]). Destructive read defenses only allow code execution. Any attempt to read code pages terminates a process. In this way, destructive reads destroy the availability of gadgets to attackers. However, destructive read defenses cannot completely eliminate all gadgets. For example, the runtime code generation capability of JIT compilers allows the creation of multiple copies of the same code (e.g., two native code regions can be created from the same JavaScript code, one copy is used for disclosing layout, and another copy is used for mounting attacks). Besides, loading and unloading features of dynamic libraries allow attackers to load, disclose, destroy, and unload code pages. A fresh loading of the destroyed code pages can be used in attacks utilizing the layout information of the disclosed code pages. Similarly, attackers can infer code layout by creating new processes (e.g., creating new tabs in browsers using JavaScript) and making an informed guess about neighboring bytes after disclosing a few bytes (i.e., implicit reads [186]). Thus, JIT compilers, load/unload features, new process creation, and implicit reads allow attackers to get gadgets even in the presence of destructive read defenses.

***4) Code Pointer Integrity (CPI)***. CPI defenses aim to ensure the integrity of code pointers either

from leaking through disclosure attacks or from modifying by attackers. To do so, code pointer obfuscation techniques identify sensitive memory objects that could lead to code pointers, encrypt the sensitive memory objects [49, 126, 130], store the memory objects in a translation table [12], and hide the memory objects in a secure and isolated memory region [115, 126]. Table 2.2 shows a summary of CPI-based leakage-resilient defenses. CPI is a powerful defense but there are few attacks (e.g., Isomeron [62] and COOP [169]) that have demonstrated their capability to bypass some CPI-based defenses.

***5) Data Pointer Integrity (DPI)*** Recent attention on non-control-oriented or data-only attacks [101, 105] motivated researchers to develop practical Data-Flow-Integrity (DFI) [29] solutions (details of non-control attacks in [40]). Currently, it is challenging to implement a practical DFI solution considering the overhead of data-flow tracking.

### 2.1.3 Control-Flow Integrity

Completely orthogonal to defenses like ASLR, DEP, stack cookies, and leakage-resilient solutions, Control Flow Integrity (CFI) [2] has gained interest due to its capability to prevent all control-oriented attacks in its ideal form. However, the requirement of source code/debug information and expensive performance overhead restrict CFI from being practical. In practice, imprecision in resolving indirect control-flow transfers impacts CFI's security guarantees. Besides, there are trade-offs between using coarse-grained CFI (performance overhead is low when enforced) and precise CFI (performance overhead is high when enforced). To make CFI fast and practical, many researchers have focused on developing the coarse-grained version of CFI [54, 83, 130, 132, 140, 222, 224]. Table 2.3 shows several practical CFI defenses with their costs.

Even with the great promise of CFI for protecting control-oriented attacks, attackers may find ways to launch new exploits such as control-oriented [46, 75, 86, 169] and non-control-oriented [27, 101, 105] exploits as demonstrated before, where the exploits conform with CFI. The latest advancement in control-flow transfers such as MLTA [124] significantly advances CFI

that can prevent most control-oriented attacks.

Table 2.3: Practical CFI solutions with their average performance overhead

| **CFI Tools** | **bin-CFI** [224] | **CCFIR** [222] | **CFL** [19] | **KCoFI** [54] | **XFI** [70] | **CCFI** [130] | **O-CFI** [132] | **MCFI** [140] | **RockJIT** [141] | **Lockdown** [152] |
|---|---|---|---|---|---|---|---|---|---|---|
| Avg. Performance Overhead | 8.54% | 3.6% | 4.5% | 13-27% | 5-10% | 3-18% | 4.7% | 5% | 14.6% | 19.09% |

### 2.1.4 Powerful and Influential Attacks

The deployment of defenses like stack canaries, W⊕X, ASLR, fine-grained ASLR, CPI, execute-only memory (XOM), and CFI is not enough for some advanced attackers because some attackers have proved their capabilities to bypass many modern defenses. For example, attackers have demonstrated advanced exploits such as JIT-ROP [185], CROP [81], BROP [18], PIROP [89], COOP [169], AOCR [163], ZombieG [186], CFB [27] and many more on the machines that are equipped with the modern defenses. Table 2.4 shows these attacks along with their techniques and capability.

Snow et al. demonstrated a Just-In-Time Code Reuse attack (JIT-ROP) that can bypass the fine-grained ASLR defense [108, 151, 209] by leveraging a memory corruption vulnerability (e.g., heap overflows, use-after-free, etc.) [185]. The attack is particularly strong because it starts with an information leak and ends with generating an exploit payload by just-in-time compiling a custom attack program.

Bittau et al. demonstrated that a remote unknown binary (equipped with the canary, NX, and ASLR) can be exploited using a stack vulnerability and a 1-bit information leakage if the binary restarts from a crash and does not re-randomize its address space [18]. More recently, Gawlik et al. showed that the ASLR defense can be bypassed without the information leakage for fault-tolerant programs [81]. They developed a memory probing technique (so-called memory oracles) to search for reference-less sensitive data (e.g., Process Environment Block (PEB), Thread Environment Block (TEB), etc. that contain addresses of the mapped modules, stack boundaries, and exception

Table 2.4: Powerful and influential attacks demonstrated to bypass different countermeasures such as coarse-grained ASLR, fine-grained ASLR, CPI, Destructive read (i.e., execute-only memory), and CFI.

| **Attack** | **Technique Used** | **Ability to Bypass** |
|---|---|---|
| JIT-ROP [185] | - ROP gadgets<br>- Repeated memory disclosures using call and jump links | - Fine-grained ASLR<br>- Partial CPI<br>- Partial destructive read |
| BROP [18] | - Information about whether a process crashes upon receiving an input<br>- Stack reading to read stack canaries<br>- Remote ROP gadget lookup | - Coarse-grained ASLR |
| CROP [81] | - Crash-resistance ability of applications<br>- No memory disclosure<br>- Memory probing technique<br>- Exploit by chaining functions | - Fine-grained ASLR<br>- CPI |
| Isomeron [62] | - virtual table pointers for repeated memory disclosures | - Fine-grained ASLR<br>- Partial CPI |
| EHH [28] | - Combination of ROP, JOP, and COP gadgets | - ROPecker and kBouncher |
| PIROP [89] | - No memory disclosure<br>- Partial pointer overwrite to overwrite LSB of a pointer<br>- ROP gadgets with relative offsets | - ASLR upto page-level granularity |
| CFB [27] | - Printf-oriented programming<br>- Controlling arguments of printf() to achieve Turing-complete computation | - Fine-grained CFI |
| BATE [17] | - Pop-ret ROP gadgets<br>- Spiller JOP gadgets | - Coarse-grained CFI |
| StackDefiler [46] | - Exploiting user-mode return addresses | - fine-grained CFI |
| OOC [86] | - Entry-point and call-site gadgets | - Coarse-grained ASLR<br>- Coarse-grained CFI |
| TROP [75] | - Typed ROP or TROP gadgets | - Runtime type checking based CFI |
| COOP [169] | - Virtual functions as gadgets | - Fine-grained CFI |
| AOCR [163] | - Profiling indirect code pointers by observing the execution state of diversified code space | - CPI |
| MTP [73] | - Overwriting data pointers to launch timing side channel<br>- Timing side channel leaks information about safe memory region | - CPI |
| ZombieG [186] | - Zombie gadgets from JIT compilers, load/unload features, new process creation, and implicit reads | - Destructive read |

handlers) in a program's address space. Once the sensitive data is read, attackers can perform attacks by leveraging functions (or addresses) from the sensitive data.

Many of these advanced attacks in Table 2.4 require assumptions and specific environments. For example, AOCR [163] observes indirect code pointers (specially for library functions, e.g., `open()`, `write()`, etc.) in stack as a starting pointer to identify and chain AOCR gadgets and perform position-independent code reuse. This indirectly means the necessity of enough library code pointers in a stack. For CROP [81], a requirement is crash-resistance, i.e., an application should progress even though the application produces memory corruptions and access faults. A significant portion of the BROP attack [18] depends on the **P**rocedural **L**inkage **T**able (PLT) entries. The attack also requires restarting a process after a crash using `fork` instead of `execve` for a PIE-binary.

## 2.2 Data-Oriented Attacks

The code reuse attacks dominated in the last decade due to their capability of bypassing DEP or NX. However, researchers have put significant effort into developing practical security solutions for preventing code-reuse attacks. As discussed above, the solutions are broadly in five categories: *i)* fine-grained address space randomization (ASR [84], ASLP [108], CCR [111], Selfrando [47], etc.), *ii)* re-randomization (*e.g.*, TASR [16], Shuffler [216], Remix [38], ASLR-Guard [126], etc.), *iii)* memory leakage prevention (*e.g.*, ASLR-Guard [126], XnR [11], Readactor [52], Heisenbyte [197], etc.), *iv)* code pointer integrity (*e.g.*, CPI [114], PointGuard [51], etc.), and *v)* CFI (*e.g.*, BCFT [83], CCFIR [222], bin-CFI [224], etc.).

With the advances toward practical code pointer protection countermeasures [12, 49, 103, 104, 115, 126, 130, 158] and practical Control-Flow Integrity (CFI) [24, 83, 124, 130, 132, 152, 222, 224], we anticipate a shift towards the use of data object/pointer-manipulation as the attack vector as the manipulation works in the presence of code pointer protection and CFI countermeasures. This is why in recent years, we observed a momentum in data-oriented attacks (also known as non-control attacks) [100, 101, 106, 133, 162, 168, 196, 219] even though data-oriented attacks were

introduced more than a decade ago [35].

## 2.3 Techniques for Protecting Object/Pointer Integrity

One fundamental requirement of the many abovementioned advanced attacks is memory disclosure or information leak. Oftentimes, an attacker uses vulnerabilities in an application's memory (e.g., user-after-free, type confusion, etc.) and weaknesses in system internals (e.g., vulnerabilities in Glibc malloc implementation, Heap Feng Shui [190], etc.) to leak memory contents [9, 82, 96, 190, 192, 196], particular code and data pointers. For example, JIT-ROP [185] requires a code pointer leak. On the other hand, some attacks require data pointers [73, 173]. That is why we have observed Code Pointer Integrity (CPI) solutions [12, 49, 115, 126, 130] and Data Pointer Integrity (DPI) solutions [29] for ensuring Pointer authenticity/integrity.

### 2.3.1 Pointer Integrity

Pointer integrity aims to ensure the validity of pointers, *i.e.*, the value of a pointer (the address of the target object) is not arbitrarily controllable by an attacker, even in the presence of memory corruption vulnerabilities that may allow a manipulation over the pointer value. *PointGuard* [51] encrypts all pointers at runtime by XORing them against a key generated at program initialization. The encryption on each pointer must be reversed before dereferencing a pointer. PointGuard incurs a small to medium overhead (0%∼20%), but is vulnerable to information disclosure, *e.g.*, if an attacker learns the key or the XORed ciphertext of a pointer to a known address. *Code-Pointer Integrity* (CPI) [115, 210] provides control-flow hijacking protection rather than the complete memory safety. Therefore, it incurs a very low performance overhead with around 1.9% (C program) or 8.4% (C/C++ program) slowdown. Kuznetsov *et al.* [115, 210] also introduced a relaxation of CPI with better performance properties, called code-pointer separation (CPS), to achieve better security-to-overhead trade-off. However, this solution only protects code pointers with non-control

data unchecked.

### 2.3.2 Pointer Authentication (PA)

PA [158] is a hardware pointer authenticity primitive introduced in the ARMv8.3-A processor architecture to protect programs from exploiting memory vulnerabilities. PA introduces a set of new instructions for calculating and verifying a *Pointer Authentication Code* (PAC) for pointers. The use of an unauthenticated pointer would cause a memory translation fault. Each PAC is generated using a key from a set of five different keys and a modifier. The kernel generates the five keys for each process and stores them in internal CPU registers which are not accessible from userspace code. These keys remain the same throughout the process's lifetime. Out of the five keys, two are used for generating PACs for code pointers, two for data pointers, and one for general purpose uses. The modifier usually captures the contexts of pointer declarations and accesses. To store PACs, PA uses the unused bits in the virtual address of 64-bit address space. In a 64-bit Linux kernel, PA uses 24 bits for the PACs, but the size can vary based on memory scheme and address tag usages.

However, PA has a few concerns regarding the PAC generation. Since PAC generation keys stay the same for the lifecycle of a process, and modifiers may have repeatability, attackers may reuse the previously generated PAC and pointer pair at a later stage to replace another PAC and pointer that uses the same modifier [121]. If the modifier does not uniquely capture the context, it might repeat in different contexts and allow such reuse attacks. For example, in return address signing using the stack pointer (SP) values as modifiers, a return address authenticated for one function can be used for another function if the SP values in both functions are the same.

To address these concerns, the Pointer Authentication Run-Time Safety (PARTS [121]) technique augments the PA-based defense approach and compartmentalizes the PAC generations for different pointers in different contexts. The key idea of the PARTS approach is to utilize a pointer's type as a modifier. The type potentially captures the context in which the pointer is created and dereferenced.

PARTS provides a proof-of-concept implementation based on LLVM and incurs an overhead of less than 20%.

### 2.3.3 Memory Safety Defenses

*Memory-unsafe* languages such as C/C++ lack built-in memory safety guarantees, hence memory errors are prevalent in programs written in these languages. Nevertheless, C and C++ are still widely used programming languages today [76]. Despite considerable prior research in retrofitting memory-unsafe programs with memory safety guarantees, memory-safety problems persist due to a trade-off between effectiveness and efficiency: approaches with low-overhead usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance-overhead or provide limited backward compatibility [183, 196]. *SoftBound* [134] and *Hard-Bound* [66] perform data pointer safety by associating a lower and upper bound with each data pointer, and verify the bound against metadata stored in shadow memory at runtime for C programs. *SoftBound* incurs an average performance overhead of 67% due to software-based bound check while *HardBound* performs the check using hardware logic that lowers the overhead to 9% on average. *Fat-pointer* schemes store the associated bounds metadata [113] together with pointers, *e.g.*, by increasing their length [137] or by borrowing unused bits from pointers [113]. Re-purposing parts of a pointer to store validation data has the advantage of enabling fast retrieval of pointer metadata without a need for lookups from disjoint memory. But it changes the representation of pointers in memory in ways that break both binary and source code compatibility. Fat-pointers have primarily been deployed in clean-slate ISA designs [116], and memory-safe programming languages, *e.g.*, Cyclone [56] and Rust [200]. BIMA [116] is a hardware-assisted fat-pointer scheme for the SAFE secure computing platform [166]. BIMA limits the virtual addresses to 46 bits and restricts pointer alignment to powers of two. This frees 18 bits in 64-bit pointers for encoding bounds information. BIMA demonstrates that on a clean-slate ISA design, fat pointers can be realized without a performance penalty, and a 3% memory overhead due to

segmentation caused by alignment restrictions on BIMA pointers. *Low-fat-pointers* [68, 69] are an alternative to fat pointers compatible with commodity 64-bit hardware architectures, such as x86-64. Low-fat-pointers require customized stack and heap allocators that restrict both stack frame and heap memory allocation sizes to a fixed finite set and split the main program stack and heap into several sub-stacks and sub-heaps, one for each possible allocation size. Pointer accesses are then validated according to the allocation bounds associated with the corresponding sub-stack or sub-heap. The improved compatibility comes at the cost of accuracy, as low-fat-pointers accesses are only enforced at allocation bounds. On average, low-fat-pointers adds a performance penalty of 54% (16% for out-of-bounds writes) and a memory overhead of 15% for stack data and incurs a 56% performance (13% for out-of-bounds writes) and 11% memory overhead for heap data.

Unfortunately, software-based memory safety protection incurs a significant amount of overhead, ranging from 48% to 116% [68, 69, 134, 135, 136]. Data-Flow Integrity also protects memory safety issues but incurs overhead ranging from 44% to 103% [29, 187]. On the other hand, PointGuard [49] has relatively low overhead (up to 20%) for protecting pointers, but PointGuard's memory-related assumption that attackers cannot read arbitrary memory is no longer practical. Thus, software-based pointer protection, in general, is not practical due to a high runtime overhead.

### 2.3.4 Hardware-based Defenses

Hardware-based solutions can reduce the overhead significantly. *HardBound* [66] can lower the overhead to 9% on average for pointers in C programs with architectural support. Intel's Memory Protection Extensions (MPX) incurs an average performance overhead of 50% due to the complexity of storing and loading bounds metadata. Fortunately, the ARM *pointer authentication* (PA) [158] offers a low cost (19.5% overhead for data pointer authentication [121]) and near-practical pointer authentication in the ARMv8-A processor architecture.

## 2.4 Sensitive/Critical Data Protection

The idea of protecting sensitive or critical data is not new. Palit et al. designed a compiler-level defense that protects critical data [148, 149]. However, they manually annotate the sensitive data. Similarly, FlowStitch [100] performed the automation of data-oriented attacks using predefined critical data. Our work complements this work by identifying and prioritizing the sensitive data automatically. A few automated techniques [106, 133] also determined the critical data. For example, Jia et al. [106] determined the decision-making data by recording the execution of two traces with normal execution and violated execution, and observing the data that get modified and change executions. Access-driven trace data [133] are also useful to determine and understand the critical data and their structures. However, these works are not scalable as we need huge and relevant execution and access traces. On the other hand, Pathfinder [162] can automatically navigate to sensitive data from a leaked data pointer. However, it does not indicate how to determine or label sensitive data.

# Chapter 3

# Quantifying (Re-)Randomization Security and Timing under JIT-ROP

## 3.1 Introduction

Just-in-time return-oriented programming (JIT-ROP) (e.g., [185]) is a powerful attack technique that enables one to reuse code even under fine-grained address space layout randomization (ASLR). Fine-grained ASLR, also known as fine-grained code randomization or code diversification, reorders and relocates program elements. Fine-grained randomization would defeat conventional ROP code reuse attacks [178], as the attacker no longer has direct access to the code pages of the victim program and its libraries. In other words, a leaked pointer only unlocks a small portion of the code region under fine-grained code randomization, seriously limiting the attacker's ability to harvest code for ROP gadget purposes.

JIT-ROP attacks can discover new code pages dynamically [185], by leveraging control-flow transfer instructions, such as *call* and *jmp*. Under fine-grained code randomization, the execution of a JIT-ROP attack is complex, as code page discovery has to be performed at runtime. From the defense perspective, re-randomization techniques (TASR [16], Shuffler [216], Remix [38], CodeArmor [36], RuntimeASLR [125], and Stabilizer [55]) have the potential to defeat JIT-ROP attacks. Besides, protections related to memory permission such as XnR [11], NEAR [215], Readactor [52], destructive read such as Heisenbyte [197], and pointer indirection such as Oxymoron [12] specifically aim to thwart JIT-ROP attacks. Precise implementation of Control-Flow Integrity (CFI)

can protect applications from all control-oriented attacks. The recent Multi-Layer Type Analysis (MLTA) [124] technique improves CFI precision greatly by improving the accuracy in identifying indirect call targets.

Even though the great promise of CFI for protecting control-oriented attacks, attackers may find ways to launch new exploits such as control-oriented [46, 75, 86, 169] and non-control-oriented [27, 101, 105] exploits as demonstrated before, where the exploits conform with CFI. A prime requirement of many of these exploits is information or pointer leakage. Thus, a measurement mechanism aiding the design of risk heuristics-based pointer selection and prioritization techniques is necessary for protecting pointers from leakage. Besides, from a **defense-in-depth** perspective, a critical system requires to deploy multiple complementary security defenses in practice. A single defense may fail due to deployment issues such as implementation flaws or configuration issues. Thus, despite the strong security guarantees of CFI, our ASLR investigation is still extremely necessary.

Re-randomization techniques continuously shuffle the address space at runtime. This continuous shuffling breaks the runtime code discovery process by making the already discovered code pages obsolete. However, the interval between two consecutive randomizations must satisfy both performance and security guarantees.

Quantitative evaluation of how code (re-)randomization impacts code reuse attacks, e.g., in terms of interval choices, gadget availability, gadget convergence, and speed of convergence has not been reported. We define *gadget convergence* as the attack stage where an attacker has collected all the necessary gadgets. For example, if an attacker has found at least one gadget for each type of Turing-complete (TC) operations, then the gadget set is TC convergence. TC operations include memory, assignment, arithmetic, logic, control flow, function call, and system call [161].

(Re-)randomization techniques make it difficult for current gadget finding techniques to discover all gadgets. Thus, in-depth and systematic measurement is necessary, which can provide new insights on the impact of code (re-)randomization on various attack elements, such as code pointer

leakage, various gadget sets, and gadget chain formation. It is also important to investigate how to systematically compute an effective re-randomization interval. Current re-randomization literature does not provide a concrete methodology for experimentally computing an upper bound of re-randomization intervals. Shorter intervals (e.g., millisecond-level) incur runtime overhead whereas longer intervals (e.g., second-level) give attackers more time to launch exploits. An upper bound would help guide defenders to make informed interval choices.

We report our experimental findings on re-randomization interval choices considering the speed of gadget convergence, code pointer leakage, gadget availability, and gadget chain formation, under fine-grained ASLR and re-randomization schemes.

Launching exploits is not a feasible measurement methodology to evaluate ASLR's effectiveness, due to *i)* low scalability – exploit payload is not platform or application portable, *ii)* failure to exploit may not necessarily mean security, and *iii)* low reproducibility. Our evaluation involves up to 20 applications, including 6 browsers, 1 browser engine, and 25 dynamic libraries.

We designed a measurement mechanism that allows us to perform JIT-ROP's code page discovery in a scalable fashion. This mechanism enables us to compare results from a number of programs and libraries under multiple ASLR conditions (coarse-grained, fine-grained function level, fine-grained basic block level, fine-grained instruction level, and register level). Our key experimental findings and technical contributions are summarized as follows.

- We provide a methodology to compute the upper bound $\mathcal{T}$ for re-randomization intervals. If the re-randomization interval is less than $\mathcal{T}$, then a JIT-ROP attacker is unable to obtain various gadget sets such as the Turing complete gadget set, priority gadget set, MOV TC gadget set, and gadgets from real-world payloads (see the definitions of gadget sets in Section 3.2). We compute the upper bound $\mathcal{T}$ by measuring the minimum time for an attacker to find a specific gadget set, i.e., the shortest time to reach gadget convergence for the gadget set. The upper bound ranges from 1.5 to 3.5 seconds in our tested applications such as *nginx*, *proftpd*, *firefox*, etc.

- Our findings show that starting code pointers do not have any impact (i.e., zero standard deviations) on the reachability from one code page to another. Every code pointer leak is equally viable for revealing an address space layout, suggesting that attackers' discovered gadgets eventually converge to a gadget set no matter where the starting pointer is.

- Our findings also show that the starting code pointers have an impact on the speed of convergence. That means the time needed for a JIT-ROP attacker to discover a gadget set varies with the locations of starting code pointers. In our experiments, the time for obtaining the Turing-complete gadget set ranges from 2.2 to 5.8 seconds.

- We also present a general methodology for quantifying the number of JIT-ROP gadgets. Our results show that a single-round instruction-level randomization scheme can limit the availability of gadgets up to 90% and break the Turing-complete operations of JIT-ROP payloads. Also, fine-grained randomization slightly degrades the gadget quality, in terms of register-level corruption. A stack has a higher risk of revealing dynamic libraries than a heap or data segment because our experiments show that stacks contain 16 more *libc* pointers than heaps or data segments on average.

Besides, we distill common attack operations in existing ASLR-bypassing ROP attacks (e.g., [18, 28, 62, 185]) and present a generalized attack workflow that captures the tasks and goals. This workflow is useful beyond this specific measurement study.

## 3.2 Threat Model and Definitions

Coarse-grained ASLR (or traditionally known as only ASLR [199]) randomly relocates shared libraries, stack, and heap, but does not effectively relocate the main executable of a process. This defense only ensures the relocation of the base address of a segment or module. The internal layout of a segment or module remains unchanged. The **P**osition **I**ndependent **E**xecutable (PIE) option

allows to relocate the main executable in random locations in each run. For comparison purposes, we performed experiments on coarse-grained ASLR with PIE enabled on a 64-bit Linux system.

Fine-grained ASLR, aka fine-grained code randomization or code diversification, relocates all the segments and dependencies of the main executable of a process and restructures the internal layouts of the segments. The granularity of the randomization varies, e.g., at the level of functions [47, 84, 108], basic blocks [38, 111, 213], instructions [97], or machine registers [52, 98]. We evaluated randomization schemes at various levels of granularities using Zipr[1] [95], Selfrando[2] (SR) [47], Compiler-assisted Code Randomization[3] (CCR) [111], and Multicompiler[4] (MCR) [98]. We also evaluated Shuffler [216], a re-randomization tool. We are unable to test other tools due to various robustness and availability issues.

We assume standard defenses such as W⊕X and RELRO are enabled. W⊕X specifies that no address is writable and executable at the same time. RELRO stands for Relocation Read Only. It ensures that the Global Offset Table (GOT) entries are read-only. RELRO is now by default deployed on mainstream Linux distributions.

***Layered defenses.*** CFI and Code Pointer Integrity (CPI) solutions are very powerful techniques. Yet, it is still necessary for one to experimentally measure the effectiveness of various defense implementations in practice (e.g., CPI enforcement with spatial and temporal guarantees and CFI effectiveness in different granularities [24]). From a measurement perspective, it is useful and necessary to isolate various defense factors. Decoupling them helps one better understand the individual factor's security impact. Otherwise, it might be too complicated to interpret the experimental results. This is the reason we chose to focus on ASLR defenses in this work and omit other defenses (e.g., CFI [2, 54, 83, 130, 132, 140, 152, 222, 224] and CPI [12, 51, 73, 114, 115, 126]. For similar reasons, we also omit memory permission protections (e.g., XnR [11], NEAR [215],

[1] https://git.zephyr-software.com/opensrc/irdb-cookbook-examples
[2] https://github.com/immunant/selfrando
[3] https://github.com/kevinkoo001/CCR
[4] https://github.com/securesystemslab/multicompiler

Readactor [52], Heisenbyte [197], and Execute-only-Memory (XOM[5]) [120]) for this paper. We also discuss the need for measuring code pointer protection solutions under the JIT-ROP model in Section 3.6.

We assume attackers have already obtained a leaked code pointer (e.g., a function or a virtual table pointer) through remote exploitation of a vulnerability. Such an assumption is standard in existing attack demonstrations. Also, fine-grained code randomization is applied in every executable and associated library in a target system (unless specified otherwise). A JIT-ROP attacker knows nothing about the applied fine-grained randomization.

**Native vs. WebAsm vs. JavaScript version of JIT-ROP**. While the original JIT-ROP attack was demonstrated in a browser using JavaScript, the attack approach has general applicability in both native and scripting environments. Our experiments are focused on the native execution of JIT-ROP attacks. We conducted the experiments for measuring the re-randomization upper bound using the native JIT-ROP code module. The execution time of WebAssembly is within 2x of native code execution [93]; JavaScript is on average 34% slower than WebAssembly [93]. Thus, our re-randomization intervals measured using the native execution would be conservatively applicable for the scripting environments as well. Besides, JIT-ROP is not related to the JIT compilers of JavaScript (JS) engines and does not use any flaws of JIT compilers to perform a code-reuse attack, though some work [10] uses such flaws. JIT-ROP harvests gadgets from a target binary's static code, which is finely randomized; it does not harvest gadgets from dynamically generated code (e.g., scripts). Thus, JS or WebAsm versions do not make substantial differences in gadget availability.

Next, we discuss the terms Turing-complete gadget set, priority gadget set, MOV TC gadget set, re-randomization upper bound, minimum footprint gadgets, and extended footprint gadgets.

**Definition** 1. *Turing-complete gadget set refers to a set of gadgets that covers the Turing-complete operations including memory operations (i.e., load memory LM and store memory SM gadgets),*

[5]XoM is now supported natively at the hardware level on x86 systems with memory protection keys (MPK) support and Armv7-M or Armv8-M processors.

*assignments (i.e., load register LR and move register MR gadgets), arithmetic operations (i.e., arithmetic AM, arithmetic load AM-LD, and arithmetic store AM-ST gadgets), logical operations (i.e., logical gadgets), control flow (i.e., jump JMP gadgets), function calls (i.e., CALL gadgets), and system calls (i.e., syscall SYS gadgets) [161].*

**Definition** 2. *The upper bound $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$ of a re-randomization scheme $\mathcal{P}$ under a JIT-ROP attacker $\mathcal{A}$ is the maximum amount of time between two consecutive randomization rounds that prevents $\mathcal{A}$ from obtaining a Turing-complete, priority, MOV TC, or payload gadget set, i.e., for any interval $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}} < \mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$, the gadgets obtained under $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}}$ does not converge to any of the four gadget sets.*

**Extended and Minimum footprint gadgets**: A gadget is an extended footprint (EX-FP) gadget if it is an instance of the four gadget sets. An EX-FP gadget may contain additional instructions that may cause side effects in an attack payload. EX-FP gadgets include the longer memory addressing expressions. A minimum footprint (MIN-FP) gadget is also an instance of the four gadget sets without causing any side effects.

Our definition of the Turing-complete gadget set represents our best efforts, by no means the only way. For example, a pair of load (LM) and store (SM) gadgets may potentially replace a move (MR) gadget. However, they may not be directly equivalent due to possibly mismatching memory offsets of EX-FP load gadgets or the scarcity of MIN-FP load gadgets. Excluding load-n-store from the Turing-complete gadget set might underestimate attackers' capabilities, while including them might overestimate attackers' capabilities. We perform our measurements considering the Turing-complete gadget set that enables the highest expressiveness of ROP attacks. However, under this condition, our results might underestimate the attackers' capabilities. To balance an attacker's capabilities, we further break down the Turing-complete gadget set into two smaller gadget sets: *i)* priority gadget set and *ii)* MOV TC gadget set. The *priority* gadget set includes 10 most frequently used gadgets in 15 real-world ROP chains from Metasploit. The *MOV Turing-complete* gadget set [67] requires six MOV gadgets and four unique registers. Besides, we also include three real-world ROP payloads from Metasploit in our measurement.

New metrics proposed by Brown and Pande's [21] work – functional gadget set expressivity and special-purpose gadget availability – are new leads that will help relax the expressiveness condition of the Turing-complete gadget set in the future.

Our security definition of the upper bound in Definition 2 is specific to the JIT-ROP threat, and is not applicable to other threats (e.g., side-channel threats). A shorter interval may still allow attackers to gain information. However, as our Section 3.3 shows, without gadgets that information may not be sufficient for launching exploits.

## 3.3 JIT-ROP vs. Basic ROP Attacks

We manually analyze a number of advanced attacks to extract common attack elements and identify unique requirements. We illustrate the key technical differences between JIT-ROP and conventional (or basic) ROP attacks. This section helps one understand our experimental design in Section 3.4 and findings in Section 3.5. We analyze various attack demonstrations with a focus on attacks (e.g., [18, 28, 62, 185]) in our threat model.

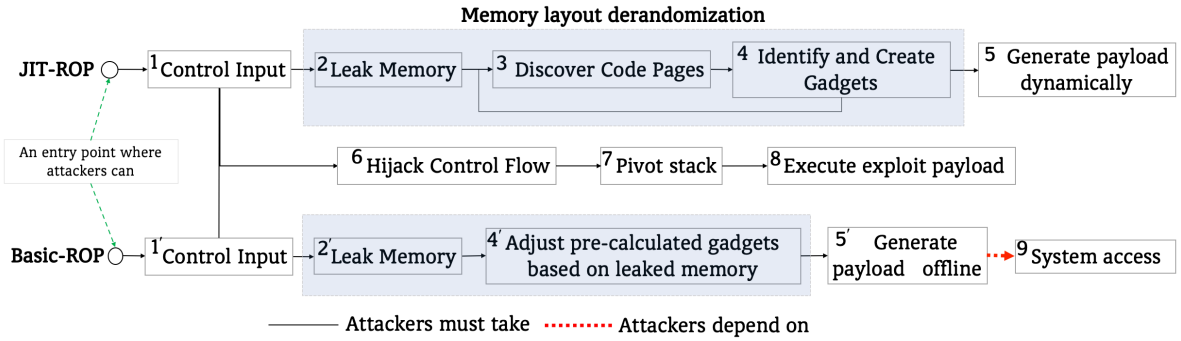



Figure 3.1: An illustration of the commonalities and differences between a conventional (or basic) ROP attack (bottom) and a JIT-ROP attack (top). The top gray-box highlights the key steps in JIT-ROP to overcome fine-grained ASLR.

To overcome both coarse- and fine-grained ASLR and conduct an attack using privileged operations, an attacker needs to perform the tasks presented in Figure 3.1. The attack workflow has three major components: ***memory layout derandomization***, ***system access***, and ***payload generation***.

### 3.3.1 Memory Layout Derandomization

Derandomizing an address space layout is the key for mounting code-reuse attacks. Due to the W⊕X defense, attackers need to derandomize the memory layout to discover gadgets (steps ②-④ for JIT-ROP and steps ②′ and ④′ for basic ROP in Figure 3.1). Usually, attackers leverage memory corruption vulnerabilities to leak memory [192] and derandomize an address space layout using the leaked memory. This step requires overcoming several obstacles.

**Memory disclosure**. The most common way of derandomizing memory layout is through a memory disclosure vulnerability. Attackers use vulnerabilities in an application's memory (e.g., heap overflows, use-after-free, type confusion, etc.) and weaknesses in system internals (e.g., vulnerabilities in the glibc malloc implementation or its variants [9, 96], Heap Feng Shui [190], and Flip Feng Shui [160]) to leak memory contents (Steps ② and ②′). Details on memory corruption can be found in [82, 196] and an example in [192].

**Code reuse**. Due to W⊕X defense, adversaries cannot inject code in their payload. ROP [178] and its variants Jump-Oriented Programming (JOP) [20] and Call-Oriented Programming (COP) [86] can defeat this defense. These techniques use short instruction sequences (i.e., gadget) from the code segments of a process' address space and allow an adversary to perform arbitrary computations. ROP tutorials can be found in [64, 185]. The difference between basic ROP [178] and JIT-ROP [185] is described next.

**Basic ROP**. Coarse-grained ASLR only randomizes the base addresses of various segments and modules of a process. The content of the segments and modules remains unchanged. Thus, it is feasible for an adversary to launch a basic ROP attack [198] using gadgets given a leaked address from the code segment of interest. The adversary only needs to adjust the addresses of pre-computed gadgets w.r.t. the leaked address. Step ④′ in Figure 3.1 is about this task.

**Just-in-time ROP**. Since adjusting the addresses of pre-computed gadgets (as in the basic ROP) no longer works under fine-grained ASLR, an attacker needs to find gadgets dynamically at the

time of an exploit. Scanning a process' address space linearly for gadgets from a disclosed code pointer may not be effective because this linear scanning may lead to crash the process due to reading an unmapped memory. A powerful technique introduced in JIT-ROP [185] is the recursive code page harvest, which is explained next.

The **recursive code harvest** technique exploits the connectivity of code in memory to derandomize and locate instructions (step ③ in Figure 3.1). The technique identifies gadgets at runtime by reading and disassembling the text segment of a process. The technique computes the page number from a disclosed code pointer and reads the entire 4K data of that page. A light-weight disassembler converts the page data into instructions. The code harvest technique searches for chain instructions, such as *call* or *jmp* instructions to find code pointers to other code pages.

An illustration is shown in Figure 3.2. The code harvest process starts from the disclosed pointer (0x11F95C4), reads 4K page data (0x11F9000-0x11F9FFF), disassembles the data, searches for *call* and *jmp* instructions to find other pointers (0x11FB410 and 0x11FCFF4) to jump to those code pages. This process is recursive and stops when all the reachable code pages are discovered.

Snow *et al.* demonstrated the JIT-ROP attack in a browser. Since exploiting a memory corruption bug remotely covers a wide variety of exploits, a browser is an ideal interface for JIT-ROP attacks. The scripting environment of a browser enables easy interfacing of a JavaScript-based JIT-ROP attack payload. Similarly, JIT-ROP attack payload can be embedded into a PDF reader that supports JavaScript (e.g., Adobe Reader). However, an attacker must convert any non-scripting attack code to script for the scripting environment. For example, the original JIT-ROP framework was written in C/C++ and was transpiled to JavaScript to demonstrate on Internet Explorer.

**Gadget identification**. In step ④ of Figure 3.1, attackers identify gadgets by scanning for byte values corresponding to *ret* opcodes (e.g., 0xC2, 0xC3) from the read code pages and perform a narrow-scoped backward disassembly. The adversary performs step ③ and ④ repeatedly to find required gadgets for the target exploit.

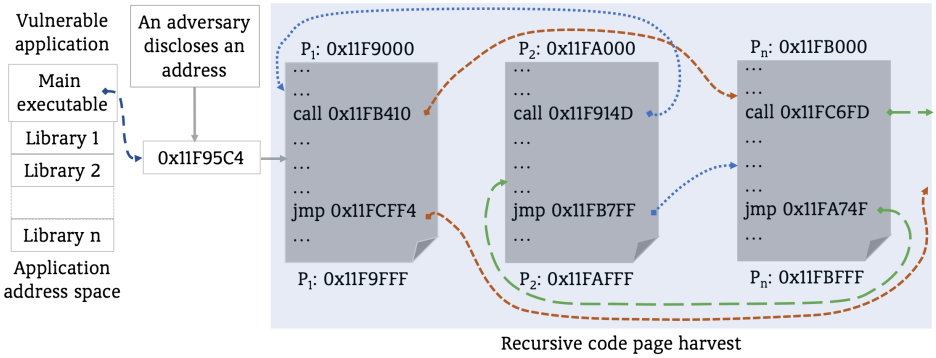



Figure 3.2: An illustration of the recursive code harvest process of JIT-ROP [185]. An adversary discloses an address from the main executable or libraries (in this case from the main executable) of an application through a vulnerability.

### 3.3.2 System Access

Attackers need to issue system APIs or gadgets to perform privileged operations. If the CFI defenses (e.g., BCFT [83], CCFIR [222] and bin-CFI [224]) are not enforced, adversaries do not need to invoke the entire functions to ensure legitimate control flow. An adversary can just chain together enough gadgets for setting up the arguments of a system call and invoking it. This observation is particularly true for Linux, which is the focus of this paper. In Windows exploits [185], the approach can be slightly different, as adversaries commonly invoke a system API instead of invoking a system call directly. *Syscall* gadgets can be found in an application's code or dynamic library. For basic ROP attacks, attackers can adjust pre-computed system gadgets from dynamic libraries, given that she manages to obtain a code pointer from a dynamic library (e.g., *libc*). Step ⑨ in Figure 3.1 is for this task. This task is performed manually and offline. The attacker may obtain the library code pointer from an application's stack or heap or data segment. One can find system gadgets through step ④ in JIT-ROP.

### 3.3.3 Payload Generation

Attackers generate payloads by putting many pieces (e.g., gadgets, functions, constants, strings, etc.) together. This process must ensure a setup for calling system APIs or system gadgets. An

attacker generates a payload dynamically at step ⑤ under fine-grained code randomization or manually at step ⑤′ under coarse-grained code randomization and stores the payload in a stack/heap. Because a payload is primarily a set of addresses that point to some existing code in an application's address space, attacks do not execute anything stored in a stack/heap, which is protected by W⊕X. An attacker may utilize the same vulnerability as in step ② or a different vulnerability to hijack a program's control flow at step ⑥ to redirect the flow to the stored payload. A payload usually targets to achieve an attack goal, e.g., memory leak or launching a malicious application/root shell.

Attack chains with minimal side effects are desirable for attackers, i.e., having a payload that fulfills attack goals without generating any unnecessary computations. However, this property may not be guaranteed if code randomization limits gadget availability. We refer to the side effects of gadgets as *footprints*. We defined the *minimum* and *extended* footprint gadgets in Section 3.2.

For ROP attacks (e.g., [28]) that bypass control-flow integrity (CFI) defenses, the attackers also need to prepare specialized payloads in addition to the previous tasks. For example, the Flashing (FS) and Terminal (TM) gadgets in Table 3.1 were designed by Carlini and Wagner [28] to bypass specific CFI implementations (namely, kBouncer [150] and ROPecker [43]).

## 3.4 Measurement Methodologies

We describe our measurement methodologies for evaluating fine-grained ASLR's impact on the memory layout derandomization, system access, and payload generation of JIT-ROP. One major challenge is how to **quantify** the impact of fine-grained code randomization. Our approach is to count the number of available gadgets under the JIT-ROP code harvest mechanism. Other challenges are how to quantify *i)* the difficulty of accessing privileged operations and *ii)* the quality of gadget chains. For the former, our approach is to measure the number of system gadgets and count *libc* pointers in a stack or heap or data-segment of an application. To quantify the quality of gadget

chains, we design a register-level measurement heuristic by computing the register corruption rate.

### 3.4.1 Methodology for Derandomization

**Gadget selection**. We manually extracted 21 types of gadgets from various attacks [18, 27, 28, 86, 185]. These gadget types include load memory (`LM`), store memory (`SM`), load register (`LR`), move register (`MR`), arithmetic (`AM`), arithmetic load (`AM-LD`), arithmetic store (`AM-ST`), `LOGIC`, jump (`JMP`), call (`CALL`), system call (`SYS`), and stack pivoting (`SP`) gadgets. In addition to these, the gadget types also include some attack-specific gadgets such as call preceding (`CP`), reflect (`RF`), call site (`CS2`) and entry point (`EP`) gadgets. Table 3.1 shows those gadget types in more detail.

These 21 types of gadgets include the Turing-complete gadget set (see Definition 1). These gadgets also include the priority and MOV TC gadget sets (Table 3.2). We use the Turing-complete, priority, and MOV TC gadget sets for our evaluation because we can precisely identify those gadgets. We also include gadgets from three real-world ROP payloads from Metasploit [58, 59] and Exploit-Database [25]. We leave the attack-specific gadgets out of our evaluation due to the lack of their concrete forms and attack goals. Attackers used the attack-specific gadgets to trick defense mechanisms. We also discuss the evaluation of the block-oriented gadgets used for Block-Oriented Programming (BOP) [105].

**Methodology for single-round randomization experiments**. In our experiments, we measure the occurrences of gadgets from the Turing-complete gadget set under fine-grained code randomization schemes. To enforce the code randomization schemes, we used four relatively new code randomization tools: Zipr [95] (instruction-level randomization), SR [47] (function-level randomization), CCR [111] (block-level randomization), and MCR [98] (function + register-level randomization), because of their reliability. Table 3.3 shows the key differences between these schemes. We compile and build a coarse- and a fine-grained version of each application or dynamic library for each run using each of the four randomization tools, i.e., each run has a different randomized code. We use LLVM Clang 3.9, Clang 3.8 and GCC 5.4 as the compilers for CCR, MCR and SR,

Table 3.1: Gadgets used in advanced ROP attacks [18, 27, 28, 86, 185] . $\bigtriangleup$ indicates an addition/subtraction/multiply/division. $\phi$ indicates logical operations such as and, or, left-shift, and right-shift. $\bigtriangledown$ indicates any operation that modifies stack pointer (SP). SN $\rightarrow$ Short name. TC? indicates whether a gadget is included in the Turing-complete gadget set or not.

| Gadget types | Purpose | Minimum footprint | Example | TC? | SN | Source |
|---|---|---|---|---|---|---|
| Move register | Sets the value of one register by another | mov reg1, reg2; ret | mov rdi, rax; ret | ✓ | MR | [185] |
| Load register | Loads a constant value to a register | pop reg; ret | pop rbx; ret | ✓ | LR | [27, 185] |
| Arithmetic | Stores an arithmetic operation's result of two register values to the first | $\bigtriangleup$ reg1, reg2; ret | add rcx, rbx; ret | ✓ | AM | [185] |
| Load memory | Loads a memory content to a register | mov reg1, [reg2]; ret | mov rax, [rdx]; ret | ✓ | LM | [27, 185] |
| Arithmetic load | $\bigtriangleup$ a memory content to/from/by a register and store in that register | $\bigtriangleup$ reg1, [reg2]; ret | add rsi, [rbp]; ret | ✓ | AM-LD | [185] |
| Store memory | Stores the value of a register in memory | mov [reg1], reg2; ret | mov [rdi], rax; ret | ✓ | SM | [185] |
| Arithmetic store | $\bigtriangleup$ a register value to/from/by a memory content and stores in that memory | $\bigtriangleup$ [reg1], reg2; ret | sub [ebx], eax; ret | ✓ | AM-ST | [185] |
| Logical | Performs logical operations | $\phi$ reg1, reg2; ret<br>$\phi$ reg1, const; ret<br>$\phi$ [reg1], reg2; ret<br>$\phi$ [reg1], const; ret | shl rax, cl; ret; | ✓ | LOGIC | [161] |
| Stack pivot | Sets the stack pointer, SP | $\bigtriangledown$ sp, reg | xchg rsp, rax | × | SP | [185] |
| Jump | Sets instruction pointer, EIP. | jmp reg | jmp rdi | ✓ | JMP | [185] |
| Call | Jumps to a function through a register or memory indirect call | call reg or call [reg] | call rdi | ✓ | CALL | [185] |
| System Call | Invokes system functions | syscall or int 0x80; ret | syscall | ✓ | SYS | [161] |
| Call preceded | Bypasses call-ret ROP defense policy | mov [reg1], reg2;<br>call reg3 | mov [rsp], rsi;<br>call rdi | × | CP | [27] |
| Context switch | Allows processes to write to Last Branch Record (LBR) to flash it | long loop. | 3dd4: dec, ecx<br>3dd5: fmul, [BC8h]<br>3ddb: jne, 3dd4 | × | CS1 | [27] |
| Flashing | Clears the history of LBR (Last Branch Record) | Any simple call preceded gadgets with a ret instruction | jmp A<br>...<br>A: mov rax, 3; ret; | × | FS | [28] |
| Terminal | Bypasses kBouncer heuristics | Any gadgets that are 20 instructions long | N/A | × | TM | [28] |
| Reflector | Allows to jump to both call-preceded or non-call-preceded gadgets | mov [reg1], reg2;<br>call reg3; ... ; jmp reg4 | mov [rsp], rsi;<br>call rdi; ... ; jmp rax | × | RF | [27] |
| Call site | This gadget chains the control to go forward when we have the control on the stack and ret | call reg or call [reg];<br>...<br>ret; | call rdi;<br>...<br>ret; | × | CS2 | [86] |
| Entry point | This gadget chains the control to go forward when we have the control of a call instruction | pop rbp;<br>...<br>call/jmp reg or<br>call/jmp [reg] | pop rbp<br>...<br>call/jmp reg or<br>call/jmp [reg] | × | EP | [86] |
| BROP | Restores all saved registers | pop rbx; pop rbp;<br>pop r12; pop r13;<br>pop r14; pop rsi;<br>pop r15; pop rdi;<br>ret; | pop rbx; pop rbp;<br>pop r12; pop r13;<br>pop r14; pop rsi;<br>pop r15; pop rdi;<br>ret; | × | BROP | [18] |
| Stop | Halts the program execution | Infinite loop | 4a833dd4: inc rax<br>3ddb: jmp 3dd4 | × | STOP | [18] |

respectively. We run, load or rewrite each application or library 100 times to reduce the impact of variability on the number of gadgets in each run or load.

We use ropper [167], an offline gadget finder tool, under coarse-grained ASLR. Under fine-grained

Table 3.2: Gadgets with gadget types in the priority and MOV TC gadget sets.

| Priority | | MOV TC | |
|---|---|---|---|
| **Type** | **Gadget** | **Type** | **Gadget** |
| LR | 1. pop reg<br>2. pop reg; pop reg | MR | 1. mov reg, reg/const |
| AM | 3. add reg, const | ST | 2. mov [reg], reg |
| LM | 4. mov reg, [reg]; ret | STCONSTEX | 3. mov [reg+offset], reg/const |
| JMP | 5. jmp reg | STCONST | 4. mov [reg], const |
| ST | 6. mov [reg], reg; ret | LM | 5. mov reg, [reg] |
| SP | 7. xchg rsp, reg | LMEX | 6. mov reg, [reg+offset] |
| LOGIC | 8. xor reg, reg<br>9. xor reg, const | SYS | 7. syscall |
| MR | 10. mov reg, reg<br>11. mov reg, const | | |
| CALL | 12. call reg<br>13. mov reg, reg, call reg | | |
| SYS | 14. syscall | | |

Table 3.3: Key differences in various randomization and re-randomization schemes evaluated.

| Tools | Randomization Scheme(s) | Randomization Time | Compiler Assistance Required? | Techniques | Performance Overhead |
|---|---|---|---|---|---|
| Shuffler [216] | Function-level re-randomization | Runtime | No | - Loads itself as a user space program<br>- Contains a separate thread for shuffling the functions continuously | 14.9% [216] |
| Zipr [95] | Instruction-level randomization | Static rewriting | No | - Reorders all instructions and generates ILR static rewrite rules<br>- Executes randomly scatter instructions using a process-level virtual machine (PVM) utilizing static rewrite rules or a fall-through map<br>- Keeps the same layout unless rewrite again | <5% [95] |
| SR [47] | Function-level randomization | Load time reorder | No | - Adds a linker wrapper that intercepts calls to the linker and asks the selfrando library to extract the necessary information to reorder functions<br>- Reorders functions every time when a binary is loaded into memory | <1% [47] |
| MCR [98] | Function- and register-level randomization | Compile & Link time reorder | Yes | - Reorders functions and machine registers during link time optimization<br>- Implements compile-time randomization but defers compilation until all translation units have been converted to bitcode<br>- Keeps the same layout unless compiled and built again | 1% [98] |
| CCR [111] | Function and block-level randomization | Installation time | Yes | - Extracts metadata during compilation<br>- Reorders functions and basic-block based on the metadata<br>- Keeps the same layout unless re-randomized again | 0.28% [111] |

ASLR, we write a tool to recreate the JIT-ROP [185] exploitation process, including code page discovery and gadget mining. Our tool can search for gadgets of a specific type. We scan the opcodes of *ret* (0xC3) and *ret xxx* (0xC2) and perform a narrow-scoped backward disassembly

from those locations to collect ROP gadgets. Similarly, we scan the opcodes of *int 0x80* (0xCD 0x80), *syscall* (0x0F 0x05), *sysenter* (0x0F 0x34) and *call gs:[10]* (0x65 xFF 0x15 0x10 0x00 0x00 0x00) for system gadgets. We consider the gadgets only from the legitimate instructions, not from instructions within overlapping instruction bytes.

**Methodology for re-randomization experiments**. For code re-randomization schemes, we attempted to use six re-randomization tools. However, some of the tools are unavailable and some have runtime and compile-time issues[6]; in the end, we were able to obtain only Shuffler [216]. To evaluate the impact of re-randomization, we take 100 consecutive address space snapshots from an application/library re-randomized by Shuffler [216]. Then, we manually analyze the address space snapshots.

The choice of re-randomization intervals is important for a re-randomization scheme. An effective re-randomization interval should hinder attackers' capabilities while ensuring performance guarantees. Our measurement methodology determines the upper bound (see definition 2) of effective re-randomization intervals by considering the fastest speed of gadget convergence, i.e., the minimum time for convergence. To measure the time of gadget convergence, we run the recursive code harvest process for an application and record the times it takes to converge to different gadget sets such as Turing-complete, priority, MOV TC, and payload gadget sets. We record the number of leaked gadget types that the code harvest process covered so far, while recording the convergence time. The code harvest terminates upon gadget convergence. We record multiple convergence times by starting the code harvesting process from multiple pointer locations to capture the variability. To select multiple starting pointers, we choose a random code pointer from each code page of an application. Choosing a single random code pointer from each code page allows us to identify all instructions and pointers on that code page.

[6]Remix [38] & CodeArmor [36] are not available. TASR [16] is not accessible for policy issues. Runtime ASLR [125] & Stabilizer [55] have run & compile time issues, respectively.

### 3.4.2 Methodology for System Access

We measure the difficulty of accessing privileged operations through the availability of system gadgets and vulnerable library pointers in a stack, heap or data-segment. For system gadgets, we compare the number of system gadgets under the coarse- and fine-grained randomization and compute the reduction in the gadget quantity. For the measurement of vulnerable pointers in a stack/heap/data-segment, we examine the overall risk associated with a stack/heap/data-segment by identifying the number of unique *libc* pointers in that stack/heap/data-segment. For the evaluation purpose, we do not exploit vulnerabilities to leak *libc* pointers from the stack/heap/data-segment. Rather, we assume that we know the address mapping of *libc* and can find the *libc* pointers through a linear scanning of the stack/heap/data-segment. We discuss the existence of *libc* pointers in popular applications in Section 3.5.6.

### 3.4.3 Methodology for Payload Generation

We focus on measuring the quality of individual gadgets to approximate the quality of a gadget chain. The quality of a set of gadgets for generating payloads is essential, as attackers need to use gadgets to set up and prepare register states. To measure the quality of individual gadgets, we perform a register corruption analysis for each gadget, which is discussed next.

Typically, a gadget contains a core instruction (other than *ret*) that serves the purpose of that gadget. For example, the core instruction of the gadget in Listing 3.1 is *mov eax, edx* and the gadget serves as a move register (MR) gadget. The core instruction is the instruction that an attacker needs. All the instructions (except *ret*) before or after the core instruction are usually unnecessary. However, these extra instructions may modify the source or destination register of a core instruction. If these extra instructions modify the registers of a core instruction, we treat the gadget as a corrupted gadget. In Listing 3.1, the instruction (*mov edx, dword ptr [rdi]*) before the core instruction modifies the source register (*edx*) of the core instruction and the instructions (*shr eax, 0x10; xor eax, edx*)

after the core instruction modify the destination register (*eax*). We identify three scenarios when core instructions get corrupted as follows:

```
mov edx, dword ptr [rdi]; mov eax, edx; shr eax, 0x2; xor eax, edx; ret;
```

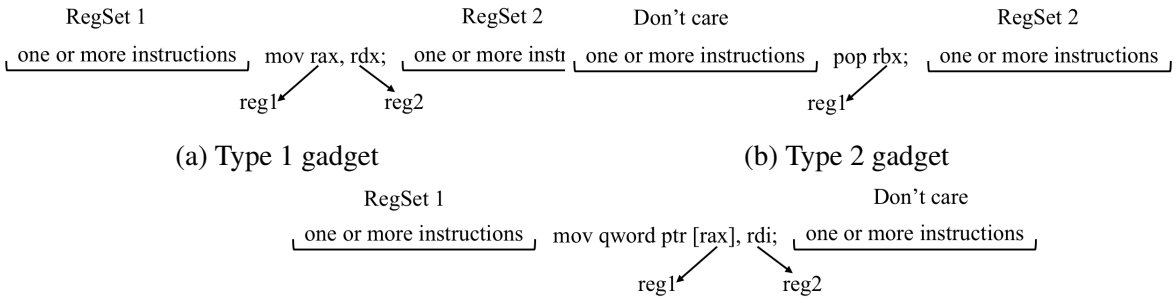
Listing 3.1: An example gadget where the core instruction is “mov eax, edx;".

***Scenario 1***: A core instruction is only affected by the instruction(s) before the core instruction,

***Scenario 2***: A core instruction is only affected by the instruction(s) after the core instruction, and

***Scenario 3***: A core instruction is affected by both the instruction(s) before or after the core instruction.

We identify three types of gadgets considering the three scenarios above where the core instructions get corrupted. Figure 3.3 shows the three types of gadgets. Each gadget has one or more instructions before or after the core instruction. For example, the Type 1 gadget in Figure 3.3 has a core instruction in the middle and one or more instructions before or after the core instruction. The core instruction has two registers for this kind. One or more instruction(s) before the core instruction may modify the source register (*rdx*) in Figure 3.3a. Similarly, one or more instruction(s) after the core instruction may modify the destination register (*rax*) in the figure.




(a) Type 1 gadget

(b) Type 2 gadget

(c) Type 3 gadget

Figure 3.3: A set of gadget types for measuring the quality of individual gadgets through the register corruption analysis

However, for the Type 2 gadget in Figure 3.3b, the core instruction has just one register. That means that the additional instructions before the core instruction cannot affect the register of the

core instruction. Thus, we do not care about the instructions before the core instruction. For Type 3 gadget in Figure 3.3c, the core instruction writes the value of *rdi* to a memory location pointed by *rax*. That is why we do not care if the register (*rax*, *rdi*) values get modified by the instructions after the core instructions.

A gadget is corrupted if registers in the core instruction get modified. We perform our register corruption analysis by identifying the corrupted registers in the core instructions of a gadget as follows.

First, we identify the set of instructions (before or after the core instruction) that can modify the source or destination register of the core instruction. We find that 17 instructions (*mov*, *lea*, *add*, *sub*, *imul*, *idiv*, *pop*, *inc*, *dec*, *xchg*, *and*, *or*, *xor*, *not*, *neg*, *shl*, and *shr*) can modify a register value of a core instruction. That means that these instructions use the source register of a core instruction as its destination register or the destination register of a core instruction as its source register. We treat the registers of such instructions as conflicting registers.

Second, we extract the conflicting registers (`RegSet1`) for Types 1 and 3 gadgets and `RegSet2` for Types 1 and 2.

Third, if the `RegSet1` and/or `RegSet2` contain more than one conflicting registers, we treat the core instruction of that gadget as corrupted, i.e., the gadget itself is corrupted.

In this way, we measure the register corruption rate for `MV`, `LR`, `AM`, `LM`, `AM-LD`, `SM`, `AM-ST`, `SP`, and `CALL` gadgets by dividing the number of corrupted gadgets by the number of all gadgets.

Next, in the following paragraphs, we discuss the code randomization and re-randomization tools briefly.

*Shuffler* [216] runs itself alongside the user space program that it aims to protect. It has a separate asynchronous thread that continuously permutes all the functions to make any memory leaks unusable as fast as possible.

*Zipr* [95] reorders the location of each instruction in an executable or library (an example in Fig-

ure 3.4). Zipr works directly on binaries or libraries with no compiler support. Zipr [95] is based on the Intermediate Representation Database (IRDB) code. Zipr shuffles code during the rewriting process, which is called block-level instruction layout randomization.

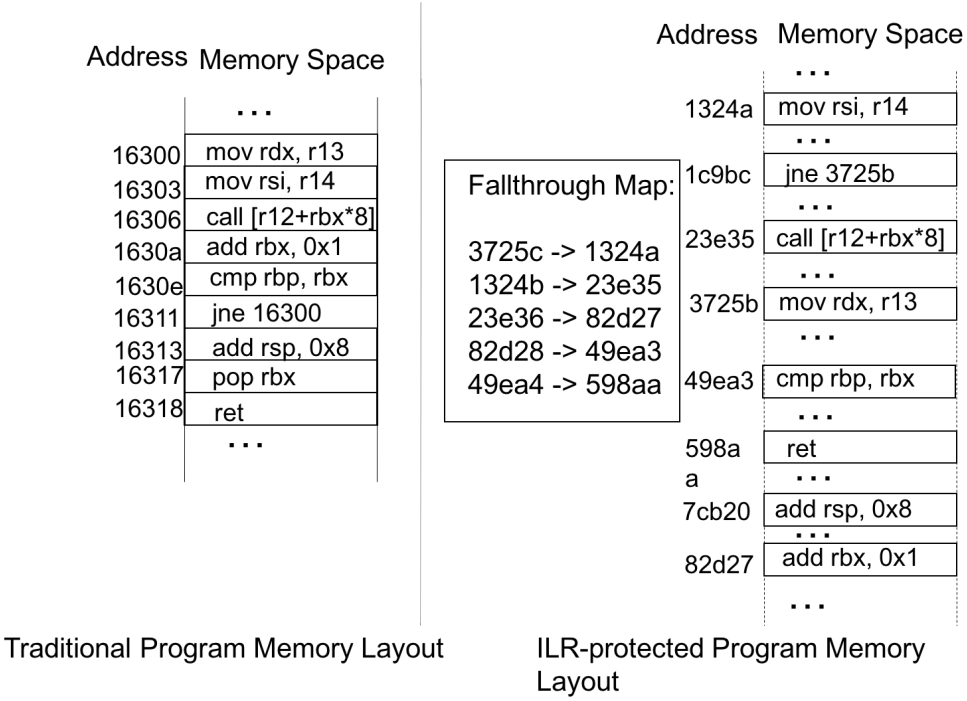



Figure 3.4: Instruction location randomization. This figure is adopted from ILR [97].

*Selfrando* (SR) [47] is compiler-agnostic and applies code diversification at the load time using function boundary-metadata called *Translation and Protection (TRaP)* and inserting a dynamic library called *libselfrando*. At the load time, *libselfrando* takes control of the execution, reorders the position of each function in an executable utilizing the TRaP information, and relinquishes the control to the original entry point of the executable.

*Multicompiler* (MCR) [52, 98] applies the code diversification at the link time. This tool randomizes functions, machine registers, stack-layout, global symbols, VTable, PLT entries, and contents of the data section. The tool also supports insertion of NOP, global padding, and padding between stack frames. We choose the function and machine register level randomization for our evaluation. MCR uses the Clang-3.8 LLVM compiler.

*Compiler-Assisted Code Randomization* (CCR) [111] applies the code diversification at the installation time, i.e., rewrites an executable binary by reordering the functions and basic blocks of the executable. This tool collects metadata for code layout, block boundaries (i.e., the basic block, functional block, and object block boundaries), fixup, and jump table of an executable during compilation and linking phases. A Python script rewrites the executable binary utilizing the collected metadata. In our experiments, CCR uses the clang-3.9 LLVM compiler.

*Availability and robustness of fine-grained ASLR tools.* We found that the majority of code diversification implementations, including ASR [84], ASLP [108], Remix [38], and STIR [213], are not publicly available. Some available tools (e.g., MCR [52, 98], CCR [111] and SR [47]) operate on the source code level that requires recompilation. We experienced multiple linking issues while using CCR and SR to compile Glibc code. The tool authors confirmed the limitations (discussed in Section 3.6). ORP [151] was the randomization tool used in Snow *et al.*'s JIT-ROP demonstration [185]. It operates on Windows binaries, incompatible with our setup.

## 3.5 Evaluation Results and Insights

***Experimental setup***. We implemented a JIT-ROP native code module. All experiments are performed on a Linux machine with Ubuntu 16.04 LTS 64-bit operating system. We write Python and bash scripts for automating our measurement process. Our code and data are available at `https://github.com/salmanyam/jitrop-native`.

We perform our experiments on the latest and stable versions of applications including *bzip2*, *cherokee*, *hiawatha*, *httpd*, *lighttpd*, *mupdf*, *nginx*, *openssl*, *proftpd*, *sqlite*, *openssh*, *thttpd*, *xpdf*, and *mupdf*, browsers including *firefox*, *chromium*[7], *tor*, *midori*, *netsurf*, and *rekonq* and browser engines such as *webkit*. We also perform our experiments on dynamic libraries. Dynamic li-

[7]Due to the incompatibility of the LLVM compiler version and the use of custom linkers with custom linking flags, we are unable to randomize the Chromium browser using SR, CCR, and MCR. Zipr also fails to randomize chromium possibly due to the large size of the executable (∼944MB). However, we include a non-randomized version of the chromium browser in our re-randomization experiments.

braries include *libcrypto*, *libgmp*, *libhogweed*, *libxcb*, *libpcre*, *libgcrypt*, *libgnutls*, *libgpg-error*, *libtasn1*, *libz*, *libnettle*, *libopenjp2*, *libopenlibm*, *libpng16*, *libtomcrypt*, *libunistring*, *libxml2*, *libmozgtk*, *libmozsandbox*, *libxul*, *libmozsqlite3*, *liblgpllibs*, *libwebkit2gtk-3.0*, and *musl*. We select these applications or dynamic libraries based on their popularity for attack demonstrations. Also, these applications or libraries are from diverse areas such as Web Server, Browser, PDF reader, networking, database, and cryptography, math, image, and system.

Table 3.4: Numbers of the applications and dynamic libraries for experiments.

| Experiment | Applications (20 Total) | Libraries (25 Total) |
|---|---|---|
| Re-randomization interval | 17 | 15 |
| Instruction-level rand. | 15 | 14 |
| Function-level rand. | 17 | 21 |
| Function + register-level rand. | 12 | 13 |
| Basic block-level rand. | 15 | 15 |

Table 3.4 shows the numbers of applications/libraries used for measuring the upper bound for re-randomization intervals and evaluating instruction-level [95], functional-level [47], function+register-level [52, 98], and basic block-level [111] randomizations. Each experiment evaluates a different set of applications and libraries because no (re-)randomization tool is capable of (re-)randomizing all of our selected applications (20 in total) and libraries (25 in total). However, we also conduct our experiments and report results using the common set of applications and libraries.

We measure a total of 11 types of gadgets for the Turing-complete set, 10 types for the priority set, and 7 types for the MOV TC set. Different payloads have different types and numbers of gadgets.

### 3.5.1 Re-randomization Upper Bound

We determine the upper bound of re-randomization intervals by measuring the fastest speed of gadget convergence across the Turing-complete, priority, MOV TC, and payload gadget sets, i.e., measuring the minimum time that an attacker needs to collect all gadget types from any of the four gadget sets. Table 3.5 shows the minimum (i.e., fastest speed) and the average time to leak all gad-

get types in a set. The minimum and the average time is calculated over 17 applications/browsers. From the table, we notice that the re-randomization upper bounds, i.e., the minimum time, range from 1.5 to 3.5 seconds. We observe some variability ($\sigma = 0.8$) in the minimum time, having the priority and MOV TC gadget sets the lowest (1.5s) and highest (3.5s) time, respectively. Intuitively, the reason for this variability could be related to the number of gadget types necessary for each gadget set. However, we observe that the minimum time for the MOV TC gadget set is larger than the TC or priority gadget set even though the MOV TC has fewer gadget types. To understand more about this variability, we analyze how gadget types are leaked over time for individual applications/browsers across the four types of gadget sets.

Table 3.5: Minimum and average time to leak all gadget types from TC, priority, MOV TC, and payload gadget sets. The percentage (%) of time is spent for leaking gadgets versus analyzing gadgets. The minimum, average, and percentages for each set are calculated using 17 applications including browsers. Payload* $\longrightarrow$ average of three payload sets.

| | Time to leak all gadget types | | Gadget analysis | |
|---|---|---|---|---|
| **Gadget set** | **Minimum (s)** | **Average (s)** | **Leak (%)** | **Analysis (%)** |
| TC | 2.2 | 4.3 | 17 | 83 |
| Priority | 1.5 | 3.5 | 13 | 87 |
| MOV TC | 3.5 | 5.3 | 16 | 84 |
| Payload* | 2.1 | 4.8 | 12 | 88 |
| Average | **2.3s** | **4.5s** | **14.5%** | **85.5%** |

Figure 3.5 shows the minimum time to obtain the Turing-complete gadget set from an individual application or browser along with a timeline for new gadget type leaks. Each gray • mark with a number $n$ on top of it represents the time to leak $n$ gadget types. The bold • mark represents the time to leak 11 gadget types from the Turing-complete gadget set. For example, it takes roughly 1 and 4.3 seconds to leak 6 and 11 gadget types, respectively from *cherokee*.

The number of leaks increases as time increases. However, the effect of the increase may not be immediate. For example, in Figure 3.5, the code harvest process takes roughly 0.7 seconds to leak 8 distinct gadget types from *netsurf*. If the time increases to 1 or 2 seconds, the number of leaked gadgets is still the same, i.e., 8 distinct gadget types. However, if the time is more than 3 seconds, the number of leaked gadgets starts to increase. We call the time between 0.7 to 3 seconds as

non-reactive.

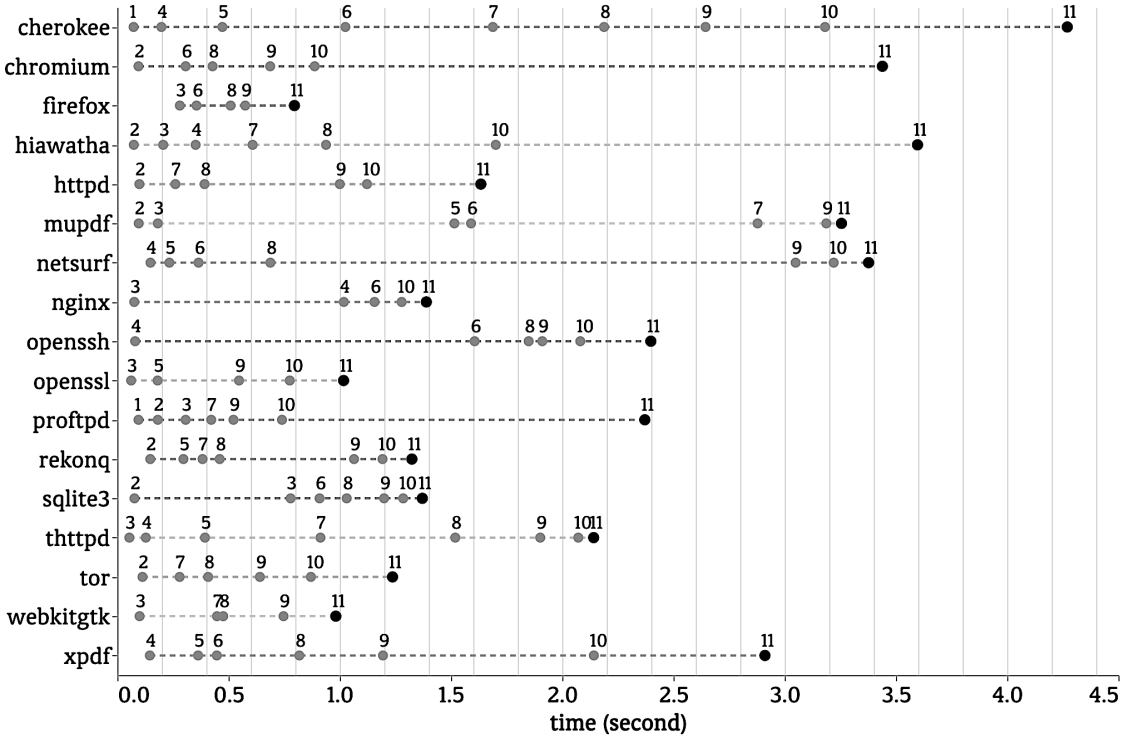



Figure 3.5: Minimum time to obtain the Turing-complete gadget set with a timeline for new gadget type leaks. Each gray filled circle (•) with a number $n$ on top of it represents the time to leak $n$ gadget types. The bold filled circle (•) indicates the time to leak all gadget types. Applications and browsers are randomized with a function-level scheme [47].

We observe a number of long non-reactive times for some other applications such as chromium (0.89–3.44s), hiawatha (1.7–3.6s), mupdf (0.18–1.52s and 1.6–2.88s), openssh (0.08–1.61s), proftpd (0.74–2.37s), and xpdf (1.19–2.14s). Most of these non-reactive times are towards the end of their timelines. These non-reactive times indicate that a few missing gadget types prevent the discovered set from being Turing-complete quickly. That is, a few types of gadgets are very scarce. The scarcest gadgets are Load-Memory (LM), Arithmetic-Load (AM-LD), and System Call (SYS) gadgets. The fundamental reason for the scarcity is that some applications (including libraries) have a few register-based memory accesses. Besides, the main executable of an application does not have SYS gadgets in most cases.

We also observe similar non-reactive times for obtaining the priority and MOV TC gadget sets. The variability in the minimum time of the four gadget sets is due to the Arithmetic-Load (AM-LD)

gadget type. Since the priority gadget set does not include AM−LD, its code page harvest process is the fastest. The time for the MOV TC gadget set is relatively longer than the TC and priority gadget set, even though MOV TC does not include AM−LD. The reason for this long time is that the MOV TC set includes several specialized Load-Memory (LM) and Store-Memory (ST) gadget types.

The MOV TC gadget set is powerful since it takes only a few *mov* instructions with four register pairs to perform the Turing-complete operations. To observe to what extent MOV TC gadgets are prevalent in applications, we count the numbers of six MOV gadgets (MR, ST, STCONSTEX, STCONST, LM, and LMEX described Table 3.2 in the Appendix) and the System Call (SYS) gadget while measuring the minimum time to find these gadgets. STCONSTEX, STCONST, and LMEX gadgets are variants of ST and LM gadgets. The average number of gadgets for MR is 51, ST is 14, STCONSTEX is 35, STCONST is 2, LM is 3, LMEX is 15, and SYS is 23. As expected, the number of Load-Memory (LM) gadgets is low, which indicates the scarcity of these gadgets. Besides, we observe the number of Store-Constant (STCONST) is also low, which is necessary for performing comparison and conditional operations.

Our re-randomization upper bound calculation includes the gadget analysis overhead. Thus, we perform additional analyses to investigate the time spent to leak address space versus gadget analysis. We find that on average around **15%** of the time is spent on leaking address space, while the rest for gadget searching (Table 3.5). This result indicates that a JIT-ROP attacker spends a significant amount of time searching for gadget types. Thus, the upper bound of re-randomization intervals is subject to change based upon an optimized gadget search strategy.

Clearly, the upper bound for the re-randomization intervals also depends on the machine (e.g., CPUs, cache size, memory, etc.) where the measurement is conducted. Using our methodology, defenders can perform the measurement on their machines to determine what intervals are appropriate for their applications, while satisfying overhead constraints. In Section 3.6, we discuss the implications of re-randomization intervals in real-world operations.

We call the upper bound of re-randomization intervals as the "best-case" re-randomization interval from a defender's perspective because the defender has to re-randomize by the time of the interval, if not sooner. This raises the question regarding the effectiveness of "best-case" intervals over "worst-case" intervals. The "worst-case" interval indicates the time required to build a useful gadget chain using a minimal set of gadgets. In reality, attackers' goals vary. It is difficult to determine a minimum set of gadgets common and necessary across all attack chains. Besides, our "best-case" interval includes the time for discovering SYS gadgets that are scarce. Some attack scenarios may not require the SYS gadgets, but the necessity of SYS gadgets or system APIs in attack chains have been shown by previous work [18, 20, 28, 62, 185].

Table 3.6: Impact of locations of pointer leaks on gadget availability. The same application has different numbers of address leaks for different schemes due to different backends (i.e., compilers) that produce different sized executables of the same program. The size of an executable is proportional to the number of code pages. Also, the numbers of gadgets from the function-level scheme [47] and function + register-level scheme [52, 98] are not comparable due to their different backends.

| | Instruction-level scheme [95] | | | Function-level scheme [47] | | | Function + register-level scheme [52, 98] | | | Block-level scheme [111] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Program | # of leaked addresses | # of MIN-FP | # of EX-FP | # of leaked addresses | # of MIN-FP | # of EX-FP | # of leaked addresses | # of MIN-FP | # of EX-FP | # of leaked addresses | # of MIN-FP | # of EX-FP |
| hiawatha | 41 | 9 | 223 | 42 | 41 | 1259 | 47 | 44 | 1042 | 39 | 31 | 793 |
| httpd | 91 | 16 | 634 | 91 | 141 | 4453 | MCR produces linking error for httpd | | | 86 | 176 | 4764 |
| lighttpd | 53 | 8 | 235 | 53 | 103 | 2512 | 68 | 118 | 2544 | 45 | 74 | 1783 |
| nginx | 114 | 26 | 788 | 121 | 222 | 5277 | 49 | 111 | 1731 | 114 | 204 | 4822 |
| proftpd | 131 | 17 | 523 | 187 | 96 | 7395 | 131 | 115 | 4466 | 131 | 125 | 3986 |
| thttpd | 10 | 8 | 172 | 17 | 22 | 583 | 16 | 31 | 535 | 15 | 24 | 428 |

### 3.5.2 Impact of the Location of Pointer Leakage

We measure the impact of pointer locations on JIT-ROP attack capabilities, by comparing the number of gadgets harvested and the time of harvest under different *starting* pointer locations. We aim to find out whether or not the number of gadgets and the time depend on the location of a pointer leakage when a fine-grained randomization scheme is applied.

**Impact of pointer locations on gadget availability**. To measure the impact of pointer locations on gadget availability, we collect the number of minimum and extended footprint gadgets by leaking

a random code pointer from **each** code page of *hiawatha*, *httpd*, *lighttpd*, *nginx*, *proftpd*, and *thttpd* and starting the code harvesting process from that leaked code pointer. Then we calculate the average number of gadgets for each leaked pointer. We leak a single code pointer from a single code page randomly because choosing any single random code pointer from a code page allows us to identify all instructions and all code pointers on that code page. Table 3.6 shows the number of leak code pointers or addresses and the numbers of minimum and extended footprint gadgets. We restrict the code harvest process to harvest gadgets from the main executable of an application to find how well the code of that application is connected. We exclude the dynamic libraries for this experiment because many applications use a common set of libraries and the gadgets from this common set of libraries (if not excluded) would dominate the total number of gadgets.

For all applications, we observe that the pointer's location does not have any impact on the total number of minimum and extended footprint gadgets. For example, regardless of the location of starting point in *nginx*, we observe 26 minimum and 788 extended gadgets when randomized by the instruction-level randomization scheme; 222 minimum and 5277 extended footprint gadgets when randomized by the function-level scheme; 111 minimum and 1731 extended footprint gadgets when randomized by function + register-level scheme; and 204 minimum and 4822 extended footprint gadgets when randomized by block-level scheme. **These findings indicate that an application's code segment is very well-connected, making JIT-ROP attacks easier**.

The numbers of leaked addresses in Table 3.6 are different for different randomization schemes because we use different backends (i.e., compilers) to enforce the schemes. Different backends optimize the same application differently. This increases/decreases the number of code pages. Since we leak a random address from each code page, the number of leaked addresses varies with tools.

**Impact of pointer locations on code harvest time**. To measure the impact of code pointer locations on the time, we measure the time required to leak all gadget types from the Turing-complete gadget set. We start the code harvest process from a random code pointer leaked from each code

page of an application or browser and record the time to collect all gadget types. Figure 3.6 shows the minimum, maximum, and average time to leak all gadgets for different applications and browsers. For a few code pointers from several applications/browsers (e.g., 3 out of 111 code pointers for *nginx* or 8 out of 40 code pointers for *openssl* or 2 out of 41 for *tor*), the code harvest process takes significantly shorter time than the average. We analyze the reason for this phenomenon.

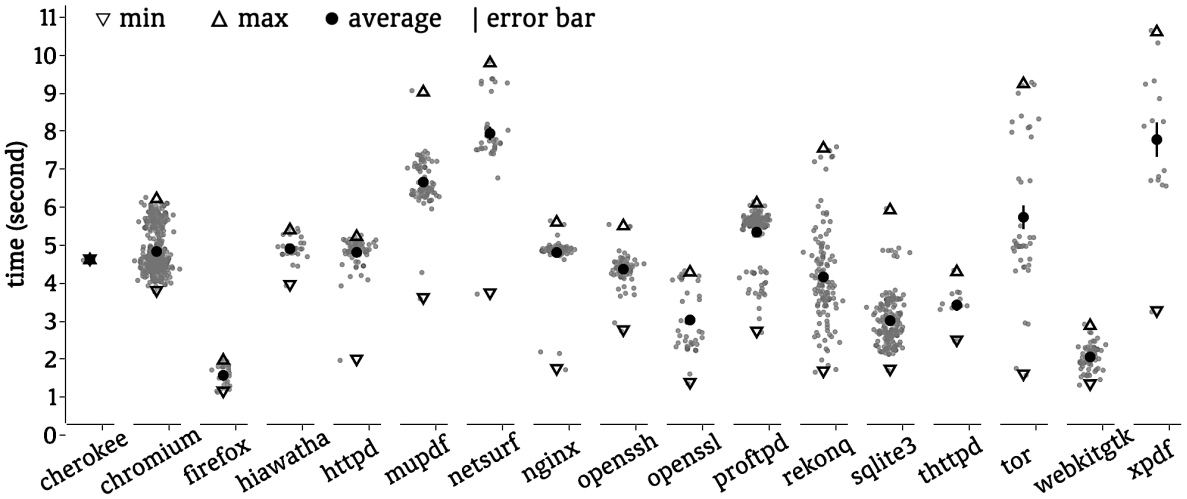



Figure 3.6: Impact of starting code pointer locations on gadget harvesting time. Each ● indicates the time for harvesting the Turing-complete gadget set. The minimum, maximum, and average time is calculated by starting code harvest process from multiple code pointer locations. A small amount of jitter has been added to the x-axis for each application/browser for better visibility of times along the y-axis.

We find that most applications/browsers have some code pages that contain a diverse set of gadgets. For example, *nginx* contains 9 code pages that have at least 5 distinct gadget types from the Turing-complete gadget set. Whenever the code harvest process accesses those code pages sooner, the discovered gadgets quickly converge to Turing-complete.

*Future directions.* Our findings imply that any valid code pointer leak is equally viable with regards to the coverage of gadgets. This observation reasserts that disrupting the code connectivity is an effective defense strategy, utilized in Oxymoron [12], Readactor [52], XnR [11], NEAR [215], Heisenbyte [197], and ASLR-Guard [126] tools. Thus, a large-scale quantitative assessment on the effectiveness of these security tools is necessary to find out the practicality and feasibility for deployment. Also, the design of risk heuristics-based pointer selection and prioritization for

protecting pointers from leakage would be an interesting direction. The idea is to prioritize code pointers based on the convergence time and data pointers based on their sensitivity (e.g., data pointers used in loops).

### 3.5.3 Impact on the Availability of Gadgets

***Impact of Single-round Randomization Schemes***. Table 3.7 summarizes the impact of fine-grained code randomization schemes on the availability of gadgets in various applications (i.e., the main executables) and dynamic libraries. We measure the numbers of the various gadgets (as mentioned in Section 3.4.1) for each application and library before and after enforcing the four fine-grained randomization schemes. We run each application or library 100 times after randomizing each time when necessary[8]. The numbers of gadgets are averaged over 100 runs for each application or library. Then the numbers of gadgets are averaged over all applications and libraries for each randomization scheme. Table 3.7 shows the overall gadget reductions in application and library categories for each randomization scheme.

On average, the number of gadgets is reduced (by 18%–28% for minimum footprint and 37%–45% for extended footprint gadgets) when applications are randomized using function-, block-, and function+register-level schemes. For dynamic libraries, the reductions range from around 21%–47% for minimum footprint gadgets and around 37%–44% for extended footprint gadgets. However, instruction-level randomization reduces the overall gadget amount significantly by around 80%–90% for both minimum and extended footprint gadgets. Table 3.7 also shows the reduction of gadgets in seven Turing-complete (TC) operations and indicates whether the Turing-complete expressiveness is preserved after applying the code randomization. The numbers before and after a vertical bar (|) indicate the reduction of minimum and extended footprint gadgets for a TC operation. Since the numbers of applications/libraries are different for different randomization schemes, we validate the schemes using a common set of applications and libraries where the result shows a

[8]One compilation with 100 runs, 100 times randomization, 100 times compilation, and 100 times rewriting are required for SR, CCR, MCR, and Zipr, respectively.

consistent reduction.

Table 3.7: Impact of fine-grained single-round randomization on the availability of gadgets in various applications and dynamic libraries. Instruction-level randomization scheme [95] is applied on 15 applications and 14 dynamic libraries, function-level scheme [47] on 17 applications and 21 dynamic libraries, function + register-level scheme [52, 98] on 12 applications and 13 dynamic libraries, and basic block-level scheme [111] on 15 applications and 15 dynamic libraries. The data of each application or library is the average result of 100 runs/loads/rewrites. The standard deviations vary between 0.3∼3.4 for minimum footprint and 5.04∼22.85 for extended footprint gadgets. ⇓ indicates reduction.

| | | | | Reduction (%) of TC gadgets in 7 TC operations (MIN-FP \| EX-FP) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Randomization schemes | Granularity | ⇓ (%) MIN-FP | ⇓ (%) EX-FP | Memory | Assignment | Arithmetic | Logical | Control Flow | Function Call | System Call | TC? |
| **Applications** | | | | | | | | | | | |
| Inst. level rando. [95] | Inst. | **79.7** | **82.5** | **97.4** \| 82.7 | 58.8 \| 81.7 | **95.9** \| 64.9 | 85.8 \| 85.4 | 49.4 \| 80.1 | 67.4 \| 83.9 | 83.3 \| 0 | ✗* |
| Func. level rando. [47] | FB | 27.63 | 36.55 | 0.8 \| 29.2 | 10.6 \| 43.5 | 19.3 \| 15.1 | 35.1 \| 35.9 | 21.1 \| 29.1 | 18.2 \| 46.9 | 0 \| 0 | ✓ |
| Func.+Reg. rando. [98] | FB+Reg. | 17.62 | 42.37 | -8.3 \| 35.0 | -5.1 \| 35.2 | 26.1 \| 44.9 | 21.3 \| 38.1 | 34.0 \| 60.2 | 11.8 \| 64.9 | 80.0 \| 0 | ✓ |
| Block level rand. [111] | BB | 19.58 | 44.64 | 5.5 \| 40.9 | 6.1 \| 47 | 26.1 \| 33.7 | 20.4 \| 37.4 | 41.2 \| 63.1 | 23.3 \| 56.3 | 0.0 \| 0 | ✓ |
| **Libraries** | | | | | | | | | | | |
| Inst. level rando. [95] | Inst. | **81.3** | **92.2** | **93.7** \| 96.1 | 60.7 \| 93 | **91.8** \| 84.9 | 84.5 \| 90.4 | 59.8 \| 93.5 | 51.8 \| 92.9 | 66.7 \| 0 | * |
| Func. level rando. [47] | FB | 46.5 | 43.8 | 24.2 \| 71.1 | 15.9 \| 31 | 41.2 \| 65.4 | 56.9 \| 25 | 34.5 \| 78.7 | 23 \| 75.8 | 3.5 \| 14.5 | ✓ |
| Func.+Reg. rando. [98] | FB+Reg. | 44.2 | 43.9 | 35.5 \| 44.8 | 35.3 \| 43.4 | 63.2 \| 61.8 | 44.8 \| 49.0 | 36.4 \| 52.1 | 43.1 \| 35.3 | 66.7 \| 0 | ✓ |
| Block level rand. [111] | BB | 20.98 | 37.0 | 7.3 \| 36.3 | 8.1 \| 32.1 | 13.9 \| 55.9 | 24.8 \| 31.6 | 22.2 \| 52.1 | 18.1 \| 44.6 | 50.0 \| 0 | ✓ |

* For instruction-level randomization scheme [95], TC is not preserved for minimum footprint gadgets, but TC is preserved for extended footprint gadgets.

The Turing-complete expressiveness of ROP gadgets is preserved in the randomized applications or libraries when the schemes are function, block, and function+register-level randomizations. However, instruction-level randomization scheme [95] does not retain the Turing-complete expressiveness for minimum footprint gadgets. The Turing-complete expressiveness is hampered when there is no gadget in one of the Turing-complete operations. For example, in Table 3.7, the reduction of minimum footprint gadgets in memory and arithmetic operations is almost 100% for applications. That means there is no gadget to do memory and arithmetic operations, which are required for reliable attacks. The reductions for libraries in the two categories (i.e., memory and arithmetic) are 93.7% and 91.8%, respectively. For both application and library cases, the reductions are not exactly 100%, because some applications/libraries contain a few gadgets. When the numbers of gadgets are averaged over all applications or libraries, the average is close to zero.

Most of the applications and libraries do not contain any *syscall* gadgets (as expected), as applications and libraries usually make syscalls through *libc*. This is why the number of *syscall* gadgets

is low (2-3) and one gadget loss leads to around 33% reduction.

We also assess the gadget availability under a *single* randomization pass of Shuffler [216] by analyzing 100 consecutive address space snapshots from *nginx* after each re-randomization with an interval of 30 seconds. On average, we observe a 24% and 3% reduction in gadget availability for minimum and extended footprint gadgets compared to a non-randomized *nginx*, respectively. The low reductions are expected, as Shuffler's security relies on continuous randomization, not a single randomization pass.

Ideally, function-level randomization does not break gadgets, only shifts the gadgets from one location to another. Basic-block or machine-register-level randomization may break some gadgets due to the memory layout perturbation and register allocation randomization. It is not surprising that the function, block, or register-level randomizations have low gadget reduction. However, instruction-level randomization perturbs the memory layout significantly as we observe a large gadget reduction by Zipr.

*Future directions.* Redefining traditional ROP gadgets into smaller (e.g., one line) building blocks and demonstrating new gadget chain compilers (e.g., two-level construction) by tackling the instruction-level perturbations are interesting new attack directions.

### 3.5.4 Impact on Performance Overhead

We measure the performance overhead of the five (re-)randomization tools to evaluate the overhead in our measurement environment. To measure the performance overhead, we use 8 applications in domains such as web servers, FTP servers, browsers, security protocols, and file compression tools. The applications are *nginx*, *httpd*, *proftpd*, *hiawatha*, *lighttpd*, *openssl*, *firefox*, and *bzip*. Applications are randomized using the five (re-)randomization tools. We use criteria such as HTTP request latency, FTP upload speed, browser page-load time, compression time, and effectiveness of cryptographic algorithms to measure the performance overhead.

We measure HTTP request latency by running an HTTP benchmark using *wrk* [85] for 30 seconds to read an HTML page from a server. The benchmark includes 12 threads and 400 HTTP open connections. To measure FTP upload speed, we run a benchmark using *ftpbench* [211]. The benchmark runs 10 concurrent operations for 10 seconds. We use OpenSSL *speed* to test the performance of aes-128-gcm, aes-256-gcm, aes-128-cbc, and aes-256-cbc algorithms. We use the Linux *time* command to measure compression time. Finally, we use a website speed test tool [203] to measure a browser's page load time. For Shuffler, we measure the overhead for three different re-randomization intervals: 10ms, 100ms, and 1s.

We run each measurement for five times and calculate the average for each application. Then, we average the overheads over the 8 applications. For Shuffler, we observe 3% overhead with 1s re-randomization interval, 5% for 100ms, and 12% for the 10ms interval consistent with the reported result [216]. We observe 23% overhead for Zipr, 10% for SR, 3% for CCR, and 10% for MCR which are comparable to or higher than what's reported. The reported overheads for Zipr, SR, CCR, and MCR are around 5% [95], 1% [47], 0.28% [111], and 1% [98], respectively.

### 3.5.5 Impact on the Quality of a Gadget Chain

The purpose of this analysis is to estimate the quality of a gadget chain. We measure the quality of a gadget through the register corruption analysis for individual gadgets, following the procedure described in Section 3.4.3. We measure the register corruption rate for `MV`, `LR`, `AM`, `LM`, `AM-LD`, `SM`, `AM-ST`, `SP`, and `CALL` gadgets. Some gadgets such as `CP`, `RF`, and `EP` (described in Table 3.1 in the Appendix) are special purpose gadgets that are used to trick defense mechanisms, such as CFI [2], kBouncer [150], and ropecker [43]. Thus, we omit these gadgets from the quality analysis.

We found that the overall register corruption rate is slightly higher ($\sim$6%) in the presence of fine-grained randomization. This slightly higher register corruption rate indicates that the formation of gadget chains is slightly harder in fine-grained randomization compared to the coarse-grained randomization.

We present the detailed results in Table 3.8, including the average number of unique registers used in each gadget. We observe the number of unique registers used in each gadget ranges from 1 to 4 in our register corruption measurement.

Table 3.8: Register corruption for various gadgets. The numbers before and after the vertical bar (|) represent the average number of unique register usage and register corruption rate in a gadget, respectively. CG → Coarse-grained. FG → Fine-grained. Fine-grained versions prepared using SR [47].

| | Program | MV | LR | AM | LM | AM-LD | SM | AM-ST | SP | CALL | Avg | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CG | Nginx | 4 \| 11% | 2 \| 0.3% | 3 \| 21% | 3 \| 44% | 3 \| 6% | 2 \| 47% | 2 \| 13% | 2 \| 6% | 2 \| 9% | — | |
| | Apache | 4 \| 16% | 2 \| 0.5% | 3 \| 37% | 2 \| 26% | 3 \| 10% | 2 \| 24% | 2 \| 5% | 2 \| 3% | 2 \| 7% | — | |
| | ProFTPD | 3 \| 69% | 2 \| 0.6% | 3 \| 7% | 2 \| 24% | 2 \| 20% | 2 \| 16% | 2 \| 11% | 4 \| 1% | 1 \| 6% | — | |
| | **Average** | **4 \| 32%** | **2 \| 0.5%** | **3 \| 21.7%** | **2 \| 31.3%** | **3 \| 12%** | **2 \| 29%** | **2 \| 9.7%** | **3 \| 3.3%** | **2 \| 7.3%** | 3 \| 16.3 | |
| FG | Nginx | 3 \| 9% | 1 \| 0.1% | 2 \| 0.1% | 3 \| 15% | 2 \| 45% | 2 \| 13% | 2 \| 47% | 1 \| 7% | 2 \| 4% | — | |
| | Apache | 3 \| 27% | 1 \| 1% | 3 \| 41% | 3 \| 27% | 2 \| 19% | 2 \| 41% | 2 \| 0% | 2 \| 2% | 3 \| 27% | — | |
| | ProFTPD | 3 \| 14% | 2 \| 1% | 3 \| 4% | 2 \| 19% | 2 \| 22% | 2 \| 35% | 2 \| 6% | 3 \| 11% | 3 \| 28% | — | |
| | **Average** | **3 \| 16.7%** | **1 \| 0.7%** | **3 \| 15%** | **3 \| 20.3%** | **2 \| 28.7%** | **2 \| 29.7%** | **2 \| 17.7%** | **2 \| 6.7%** | **3 \| 19.7%** | 2 \| 17.3 | ∼5.7%⇑ |

Sometimes, fine-grained randomization decreases the register corruption rate. For example, for Nginx, the corruption rate of the load memory (LM) gadgets is reduced from 44% to 15%, when fine-grained randomization is in place. This reduction is likely due to the relatively smaller number of gadgets in the presence of the fine-grained randomization.

*Future directions.* Designing randomization solutions to increase the register corruption rate in gadgets would be interesting as a high register corruption rate would make attacks unreliable.

### 3.5.6 Availability of Libc Pointers

This experiment measures the risks associated with an application's heap, stack, or data-segment for revealing a library location. For simplicity, we consider only the risk associated with revealing the *libc* library w.r.t. the basic ROP attacks. We count the number of unique *libc* pointers in a target application's stack, heap, and data-segment when the application reaches a certain execution point. We define the execution points for the target applications. For example, the execution point for *proftpd* is when *proftpd* is ready to accept connections. We assume that *i)* coarse-grained ran-

domization is enforced, and *ii)* adversaries cannot perform recursive code harvest to find gadgets. This experiment targets a weak attack model where an adversary leaks a (known) library pointer and adjusts pre-computed gadgets based on the leaked pointer. We regard a library pointer (e.g., *libc* pointer) as known if the pointer is loaded in the same location in the stack of an application for multiple runs. A pointer in a stack, heap, or data-segment may point to a non-library function, which in turn points to a library (e.g., *libc*).

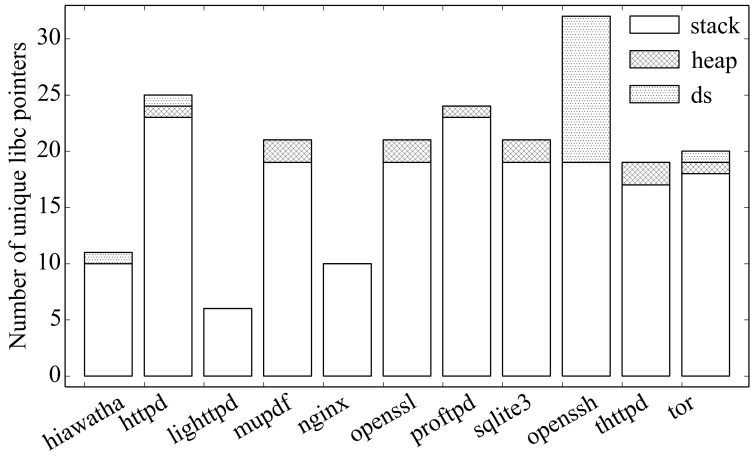



Figure 3.7: Libc pointers in the stack, heap and data segment of a program. Stacks contain more pointers, carrying higher risks of pointer leakage.

Figure 3.7 shows the number of unique *libc* pointers in the stack, heap, and data-segment of 11 applications including web servers, PDF reader, cryptography library, database, and browser. According to the observations in Figure 3.7, heap or data-segment contains only one *libc* code pointer (on average) while stack contains 17 *libc* code pointers. This finding indicates that higher risk is associated with stack than heap or data-segment. It also suggests that the safeguard and (re-)randomization of stack is more important than protecting/randomizing heap or global variables.

*Future directions.* Protecting a stack/heap/data-segment from leaking data pointers that contain code pointers is an interesting research direction. A similar risk assessment for C++ binaries may indicate the importance of protecting the read-only sections that include many pointers to virtual methods.

## 3.6 Discussion

**Metrics for evaluating fine-grained randomization**. Traditionally, both coarse-grained (e.g., PaX ASLR [199]) and fine-grained (e.g., SR [47], CCR [111], Remix [38], Binary stirring [213], ILR [97] and ASLP [108]) randomizations use entropy to measure the effectiveness of hindering code-reuse attacks. However, such an entropy measure is not useful under the JIT-ROP threat model, as chunks of code are still available. Inclusion of distances between permuted functions or basic blocks for computing entropy would not work either, because the code's semantic connectivity (e.g., through *call* and *jmp*) is still not captured. Code connectivity is what JIT-ROP attacks leverage to discover code pages. In comparison, our measurement methodology more accurately reflects JIT-ROP capabilities and is more meaningful under the JIT-ROP model. How to design an entropy-like metric to capture the degree of code isolation or the **semantic connectivity** in code is an interesting open problem.

**Availability of Block-Oriented Programming (BOP) gadgets**. We measure the numbers of BOP functional blocks for register assignments/modifications, memory reads/writes, system/library calls, and conditional jumps using the BOP compiler (BOPC) [105]. We observed almost no change in the numbers of BOP functional blocks in randomized versions compared to the non-randomized versions for CCR [111] and MCR [52, 98]. As BOPC operates on a static binary, we could not use Shuffler [216] and SR [47] because they randomize a memory layout at runtime. BOPC does not seem to run on binaries rewritten by Zipr [95].

**Impact of the compiler optimizations on gadget availability**. We assess the impact of code transformations and optimizations (-O0, -O1, -O2, -O3, -Ofast, -Os) on the availability of gadgets. We compare the unoptimized and optimized versions of *nginx*, *apache*, *proftpd*, *openssh*, and *sqlite3* to assess the impact. We find a smaller number of LM, SM, and MR gadgets in unoptimized code than optimized code. The reason is the tendency of *mov* and *ret* instructions staying together in optimized code, but not in unoptimized code. Besides, compilers sometimes emit extra instructions for optimizations that may increase gadgets.

**Reachability of gadgets**. We design our experiments based on the availability of various kinds of gadgets. However, in reality, attackers need to conduct a series of operations including finding a vulnerability or leaking memory for the actual invocations of gadgets. In Section 3.2, we assume that an attacker has already overcome the initial obstacles, especially finding a memory leak. Our experiments are focused on comparing gadget availability of various code (re-)randomization schemes using the leaked memory.

**Operational re-randomization intervals**. Our methodology helps guide software owners (e.g., server owners) to set the appropriate re-randomization intervals. For example, if the owners prioritize performance over security, they can set an interval just below $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$ (Definition 2). If the owners prioritize security over performance, they can set an interval much shorter than $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$.

**Need for randomizing Glibc**. Unfortunately, SR, CCR, MCR, and Zipr were all unable to randomize Glibc. For CCR and MCR, the LLVM Clang compiler (backend of CCR and MCR) does not have support for compiling some Glibc's GCC specific extensions such as ASM GOTO. SR also cannot randomize some parts of Glibc. That is why we evaluate a lightweight standard C library `musl-libc` [119]), but only SR can randomize `musl-libc`. Shuffler can reorder Glibc by disabling manual jump table construction.

**Key Takeaways**

❶ *Effective re-randomization upper bound.* Our methodology for measuring various gadget sets systematically by considering the gadget convergence time helps compute the effective upper bound for re-randomization intervals of a re-randomization scheme. Our results show that this upper bound ranges from 1.5 to 3.5 seconds. Applying our methodology on their machines will help re-randomization adopters to make informed configuration decisions.

❷ *All leaked pointers are created equal for gadget convergence, but not for the speed of gadget convergence*. Regardless of the location of pointer leakage, we obtained the same number of minimum and extended footprint gadgets via JIT-ROP. This observation indicates that **any** pointer leak from an application's code segment is equally useful for attackers. However, the time for

obtaining the gadgets varies for different leaked pointers.

❸ *Turing-complete operations*. Function, basic-block, or machine register level fine-grained randomization preserves Turing-complete expressive power of ROP gadgets, however, instruction-level randomization does not.

❹ *Connectivity*. Code connectivity is the main enabler of JIT-ROP. As the conventional entropy metric does not capture code connectivity, it should not be used to measure ASLR security under the JIT-ROP threat model.

❺ *Gadget quality*. Our findings suggest that current fine-grained randomizations do not impose significant gadget corruption.

## 3.7 Related Work

The related research has two themes: 1) demonstrating attacks and 2) discovering countermeasures. Attack demonstrations range from stack smashing [147], return-to-libc [112, 153, 217], to ROP [28, 31, 107], JOP [20], DOP [101], ASLR bypasses [18, 62, 81, 88, 101, 185], and CFI bypasses [17, 27, 28, 86, 105].

Researchers have also proposed a range of defenses for ROP attacks [2, 19, 32, 43, 52, 54, 63, 64, 70, 80, 87, 140, 146, 150, 151, 152, 170, 205, 222, 224], CFI bypass [222], and ASLR bypass [11, 12, 16, 38, 52, 62, 84, 97, 108, 111, 126, 128, 151, 197, 213, 215, 216]. A categorical representation of these defenses is given in our attack-path diagram (Figure 3.8). Binary analysis tools are also available to understand [181] and mitigate [206] these ROP or code-reuse attacks.

Most of the above-mentioned defenses are variants of W$\oplus$X (e.g., NEAR [215] and Heisenbyte [197]), memory safety (e.g., HardScope [143], Memcheck [138], AddressSanitizer [176], and StackArmor [37]), ASLR (e.g., fine-grained randomization [16, 38, 111, 174, 213, 216]), and CFI (e.g., CCFIR [222] and bin-CFI [224]). These defenses are capable of preventing most code-reuse

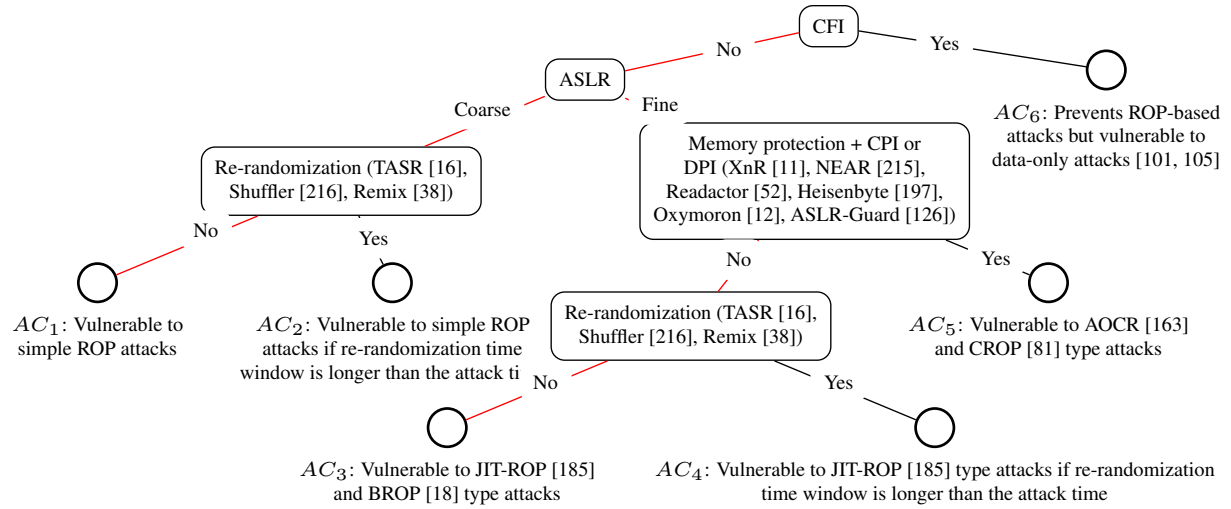



Figure 3.8: High-level view of the types of ROP attacks and attack-paths based on various security measures. Each rectangle and circle indicate security measures and attack types, respectively. AC stands for attack condition. All the attack conditions have W⊕X, PIE, Canary, and RELRO implicitly.

attacks [18, 62, 81, 185] except a few cases such as inference attacks that are performed using zombie gadgets [186] or relative address space layout [89, 163]. The latest advancement in control-flow transfers such as MLTA [124] significantly advances CFI that can prevent most control-oriented attacks. Recent attention on non-control-oriented or data-only attacks [101, 105] motivated researchers to develop practical Data-Flow-Integrity (DFI) [29] solutions (details of non-control attacks in [40]). Currently, it is challenging to implement a practical DFI solution considering the overhead of data-flow tracking.

From the defense-in-depth perspective, it is desirable to have some degree of redundancy (e.g., CFI, ASLR, or complementary solutions like anomaly detection [220]) in system protection. A single deployed defense may be compromised due to unknown implementation flaws or configuration issues. Thus, investigations in multiple directions [24, 99, 179, 205] are necessary for gauging the feasibility of existing defenses. Our work investigates various aspects of ASLR – including timing – by evaluating security metrics such as various gadget sets, interval choices, and code pointer leakages. We also assess how security tools in the ASLR domain impact on these security metrics, quantitatively.

# Chapter 4

# Data-Oriented Attacks: Exploitation Techniques and Defenses

## 4.1 Introduction

Memory-corruption vulnerabilities are one of the most common attack vectors used to compromise computer systems. Attackers exploit these vulnerabilities in different ways to perform arbitrary code execution and data manipulation. Existing memory corruption attacks are broadly two types: *i)* control-flow attacks [77, 161, 178] and *ii)* data-oriented attacks (also known as non-control attacks) [35, 100, 101, 133, 196]. Both types of attacks can cause significant damages to a victim system [27, 74].

Control-flow attacks corrupt control data (*e.g.*, return address or code pointer) in a program's memory space to divert the program's control flow, including malicious code injection [77], code reuse [153], and Return-Oriented Programming (ROP) [178]. Defenses such as stack canaries [50], Data Execution Prevention (DEP) [8], Address Space Layout Randomization (ASLR) [199], Control-Flow Integrity (CFI) [3], Intel's CET [103] and MPX [104] can prevent most control-flow attacks. In particular, CFI-based defenses [24, 83, 152, 222] have received considerable attention in the last decade. The idea is to ensure that the runtime program execution always follows a valid path in the program's Control-Flow Graph (CFG), by enforcing policies on indirect control transfer instructions (*e.g.*, ret/jmp).

In contrast to control-flow attacks, data-oriented attacks [35] change a program's benign behav-

ior by manipulating the program's non-control data (*e.g.*, a data variable/pointer which does not contain the target address for a control transfer) without violating its control-flow integrity. The attack objectives include: 1) information disclosure (*e.g.*, leaking passwords, private keys or address space layout); 2) privilege escalation (*e.g.*, by manipulating user identity data) [35]; 3) performance degradation (*e.g.*, resource wastage attack) [13]; and 4) bypassing security mitigation mechanisms [219]. As launching control-flow attacks becomes increasingly difficult due to many deployed defenses, data-oriented attacks have received much attention in the literature [100, 101, 133, 142, 168, 219].

Data-oriented attacks can be as simple as flipping a bit of a variable. However, they can be equally powerful and effective as control-flow attacks [100, 105]. For example, arbitrary code-execution attacks are possible if an attacker can corrupt parameters of system calls (*e.g.*, `execve()`) [27]. Recently, Hu *et al.* [101] have proposed Data-Oriented Programming (DOP), a systematic technique to construct expressive (*i.e.*, Turing-complete) non-control data exploits. Ispoglou *et al.* [105] also presented a code-reuse technique called Block-Oriented Programming (BOP) that utilizes basic blocks as gadgets along valid execution paths in the target binary to generate data-oriented exploits. Though data-oriented attacks have been known for a long time, the threats posed by them have not been adequately addressed due to the fact that most previous defense mechanisms focus on preventing control-flow exploits.

The motivation of this paper is to systematize the current knowledge about exploitation techniques of data-oriented attacks and the current applicable defense mechanisms. Unlike prior papers [117, 189, 196] related to memory corruption vulnerabilities, our work specifically focuses on data-oriented attacks. In addition to generic memory corruption prevention mechanisms discussed in [117, 189, 196] such as memory safety, software compartmentalization, and address/code space randomization, we mainly discuss recently proposed defenses against data-oriented attacks. Our technical contributions are as follows.

1. We systematize and categorize the current knowledge about data-oriented exploitation tech-

niques with a focus on the recent DOP attacks. We demystify the DOP exploitation technique by using the ProFTPd DOP attack [100] as a case study, and provide an intuitive and detailed explanation of this attack by analyzing its constituent steps. We discuss the automation of data-oriented attacks, *e.g.*, the Block-Oriented Programming (BOP) compiler [105], STEROIDS [154], and LIMBO [172]. We also discuss representative data-oriented exploits including their assumptions/requirements and attack capabilities (Section 4.2).

2. We present a three-stage model for data-oriented attacks and discuss recent defense techniques according to different stages. Then, we provide a comparative analysis of the approaches focusing on data-oriented attacks (Section 4.3).

3. We also discuss some open research problems and unsolved challenges (Section 7).

## 4.2 Data-oriented Attacks

In this section, we first describe why data-oriented attacks have received attention among security researchers in recent years (Section 4.2.1). Then we discuss two categories of data-oriented exploitation techniques (Section 4.2.2) and automatically generating data-oriented exploits (Section 4.2.3). We also provide a brief overview of the BOP attack technique in Section 4.2.4. We discuss the similarity and difference between DOP and BOP and the generality and practicality (difficulty) of both DOP and BOP exploits in Section 4.2.5. Then, we map representative data-oriented exploits in the literature to their assumptions/requirements and capabilities (Section 4.2.6).

### 4.2.1 Why Do Data-oriented Attacks Receive Attention?

The Data Execution Prevention (DEP) or No-Execute (NX) defense prevents stack smashing [147] and related attacks such as overwriting Structured Exception Handler. A new class of attacks, namely the code-reuse attacks such as ROP [178], dominated in the last decade due to their capa-

bility of bypassing DEP or NX. However, researchers have put significant effort to develop practical security solutions for preventing code-reuse attacks. The solutions are broadly in five categories: *i)* fine-grained address space randomization (ASR [84], ASLP [108], CCR [111], Selfrando [47], etc.), *ii)* re-randomization (*e.g.*, TASR [16], Shuffler [216], Remix [38], ASLR-Guard [126], etc.), *iii)* memory leakage prevention (*e.g.*, ASLR-Guard [126], XnR [11], Readactor [52], Heisenbyte [197], etc.), *iv)* code pointer integrity (*e.g.*, CPI [114], PointGuard [51], etc.), and *v)* CFI (*e.g.*, BCFT [83], CCFIR [222], bin-CFI [224], etc.). The enforcement of the above defenses makes most code-reuse attacks unreliable. Thus, many attackers have shifted their focus from control-oriented attacks to data-oriented attacks in recent years [101, 105].

### 4.2.2 Classification of Data-Oriented Attacks

We classify data-oriented attacks into two categories based on how attackers manipulate the non-control data in the memory space: 1) Direct Data Manipulation (DDM), and 2) Data-Oriented Programming (DOP).

*1) DDM* refers to a category of attacks in which an attacker directly/straightforwardly manipulates the target data to accomplish the malicious goal. It requires the attacker to know the precise memory address of the target non-control-data. The address or offset to a known location utilized in the attack can be derived directly from binary analysis (*e.g.*, global variable with a deterministic address) or by reusing the runtime randomized address stored in memory [100]. Several types of memory corruption vulnerabilities, *e.g.*, format string vulnerabilities, buffer overflows, integer overflows, and double free vulnerabilities [189], allow attackers to directly overwrite memory locations within the address space of a vulnerable application.

```
1 void do_authentication(char *user, ...) {
2   ...
3   int authenticated = 0;
4   ...
5   while(!authenticated){
```

```
6       type = packet_read();//Corrupt authenticated
7       /*Calls detect_attack() internally*/
8       switch(type){
9           ...
10          case SSH_CMSG_AUTH_PASSWORD:
11              if(auth_password(user, password)){
12                  authenticated = 1;
13                  break;}
14          case ...
15      }
16      if(authenticated) break;
17  }
18  do_authenticated(pw);
19  /*Perform session preparation*/
20 }
```

Listing 4.1: DDM attack in a vulnerable SSH server [35]

Chen *et al.* [35] revealed that DDM attacks can corrupt a variety of security-critical variables including user identity data, configuration data, user input data, and decision-making data, which change the program's benign behavior or cause the program to inadvertently leak sensitive data. Listing 4.1 illustrates an of attacking decision-making data in the SSH server, first reported in [35]. A local flag variable `authenticated` indicates whether a remote user has passed the authentication (line 3). An integer overflow vulnerability exists in the `detect_attack()` function, which is internally invoked whenever the `packet_read()` function is called (line 6). When the vulnerable function is invoked, an attacker can corrupt the `authenticated` variable to a non-zero value, which bypasses the user authentication (line 16). DDM allows attackers to manipulate the benign data flows in a program execution without changing its control flow. FlowStitch [100] is a technique to stitch data flows in a vulnerable program to automatically construct data-oriented exploits. It takes a pair of source (*e.g.*, private key buffer) and target (*e.g.*, a publish output buffer) in a vulnerable program as the input. The goal is to stitch the source data-flow to the target data-flow,

which could leak passwords, private keys, or randomized values, and cause privilege escalation.

There also exists multi-step DDM attacks, where an adversary exploits memory corruption vulnerabilities multiple times to write data to adversary-chosen memory locations. For example, suppose an attacker needs to change two decision-making variables while the vulnerability only allows the attacker to change one value each time. It requires a 2-step DDM. Morton *et al.* [133] recently demonstrated a multi-step DDM with Nginx (listed in Table 4.1). The attack leverages memory errors to modify global configuration data structures in web servers. Constructing a faux SSL Config struct in Nginx requires as many as 16 connections (*i.e.*, 16-step DDM) [133].

Due to the widespread deployment of ASLR, attackers may exploit DDM to infer knowledge about the address space layout of a process to bypass ASLR defenses. Data manipulation can also cause some events that result in software side-channels—typically timing side-channels—that can leak information about the address space. To infer information about an application's address space, attackers analyze the output and execution time of a portion of code. They then correlate the output and timing information by running the same portion of code locally [73, 173]. In contrast to traditional DDM attacks with direct malicious goals, *e.g.*, leaking sensitive data or modifying program behavior, inferring randomized addresses serves as the first step towards malicious goals. For example, attackers need to derandomize the address space layout before performing code reuse attacks using DOP gadgets. To do so, an attacker may overwrite a data pointer (`ptr` in Listing 4.2) to point to some byte sequences and infer knowledge about the byte sequences by observing the output, *i.e.*, the variable `result` in Listing 4.2. By pointing the data pointer to different locations of an address space and observing the output behavior, an attacker can distinguish the mapped and unmapped code pages.

```
1 struct mystruct {
2       int value;
3 };
4 void vuln_function()
5 {
```

```
6     char buf[64];
7     int result=0, length, input;
8     struct mystruct * ptr;
9     recv(socket, buf, input);
10    ptr->value = strlen(buf);
11    while (result < ptr->value) result++;
12    send(socket, &result, length);
13 }
```

Listing 4.2: Data pointer manipulation to infer knowledge about address space layout. This figure is adopted from [173].

*2) DOP* constructs expressive data-oriented exploits [101]. It allows an attacker to perform arbitrary computations in program memory by chaining the execution of short instruction sequences (referred to as *DOP gadgets*). DOP gadgets are similar to ROP gadgets that can perform arithmetic/logical, assignment, load, store, jump, and conditional jump operations. However, unlike ROP gadgets, the execution of DOP gadgets follows valid paths in a CFG. We consider Block-Oriented Programming (BOP) [105] as a form of DOP because BOP also constructs exploits using a set of gadgets (details are in Section 4.2.4). However, instead of using short instruction sequences as gadgets, BOP [105] constructs exploits by chaining basic blocks as BOP gadgets. Both DOP and BOP adhere to CFI. Without loss of generality, we use DOP to represent this exploitation technique, which misinterprets multiple gadgets and chains these gadgets together by one or more dispatchers to achieve the desired outcome. A dispatcher is a fragment of logic that chain gadgets. A typical example of a dispatcher is a loop within the influence of a memory corruption vulnerability.

Typically, a DOP attack corrupts several memory locations in a program and involves multiple steps. We differentiate the DOP exploitation technique from a multi-step DDM attack. 1) *Gadgets and code reuse.* Both DOP and BOP attack techniques involve reusing code execution through CFI-compatible gadgets. Multi-step DDM hinges on direct memory writes and does not involve

any gadget executions. 2) *Stitching mechanism and ordering constraint.* In DOP and BOP attacks, how to orderly stitch gadgets to form a meaningful attack is important. Multi-step DDM attacks, *e.g.*, crafting and sending multiple attack payloads to manipulate memory values, do not need any special stitching mechanism and thus there is no ordering constraint.

### 4.2.3 Automatically Generating Data-oriented Exploits

Research efforts have been undertaken to automate the process of generating data-oriented exploits. However, identifying how to corrupt memory values for a successful data-oriented exploit is non-trivial due to the large space of memory state configurations and attackers cannot inject malicious code of their choice. FlowStitch [100] is a tool to automatically construct DDM from memory errors. It identifies the influence range of the memory errors from the error-exhibiting trace (by triggering memory errors) and generates constraints on the program input to reach memory errors. FlowStitch then performs data-flow analysis and security-sensitive data (*e.g.*, system call parameters or configuration data) identification using benign traces, and selects stitch candidates from the identified security-sensitive data flows. It finally checks the feasibility of creating new edges with the memory errors and produces the input needed to mount a data-oriented attack. Figure 4.1(a) presents an example of a web server `wu-ftpd` with the format string vulnerability (skipped on line 5). Figure 4.1(b) shows the corresponding two-dimensional data-flow graph (2D-DFG), where numbers on the time-axis are the line numbers in Figure 4.1(a). The `seteuid(pw->pw uid)` on line 9 is intended to drop the process' root privilege. Suppose attackers want to retain root privilege by exploiting the format string vulnerability on line 5. One straightforward approach, demonstrated by Chen *et al.* [35], is to use DDM to directly modify the value of `pw_uid`. In FlowStitch, the target flow `pw->pw_uid` is first identified as a security-sensitive data flow. Then, it automatically finds another source data-flow and stitches the target flow to the source flow to achieve the same attack goal. As illustrated in Figure 4.1(b), attackers may change the base pointer `pw` to an address of a structure with a constant 0 at the offset corresponding to the `pw_uid`. The

vulnerable code then reads 0 and uses it as the argument of `seteuid`, achieving a privilege escalation attack.

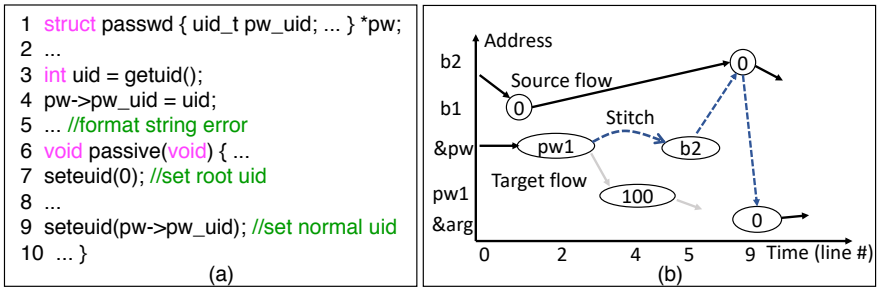



Figure 4.1: Example of FlowStitch [100].

STEROIDS [154] is a compiler which automates the process of payload preparation in a DOP exploit, but which leaves the gadget search and DOP instance setup unaddressed. However, it is unclear how practical STEROIDS is for automating the construction of end-to-end DOP exploits since it assumes DOP gadgets are arbitrarily stitchable and work for all inputs. Constructing DOP or BOP exploits still requires a lot of manual work. Ispoglou *et al.* [105] presented a Block Oriented Programming Compiler (BOPC), a mechanism to automatically evaluate a program's remaining attack surface under strong control-flow hijacking mitigations. BOPC provides an exploit programming language, called SPL, that enables defenders and software developers to define exploits independent of the target program or underlying architecture. BOPC assumes that the target binary has an arbitrary memory write vulnerability. It finds basic blocks from the target program that implement individual SPL statements, and chains these basic blocks together. Finally, BOPC simulates the BOP chain to produce a payload that implements the SPL payload.

LIMBO [172] maps the automatic exploit generation problem into the software model checking problem. Similar to software model checking, LIMBO looks for possible state transitions from an input state to the goal state, *i.e.*, the reachability from an input state to the goal state. To find the reachability, LIMBO employs concolic execution [30] with heuristics. The heuristics make the search space small by focusing on the promising paths that likely lead to the goal states. Similar to the BOPC [105], LIMBO [172] allows users to specify the input and goal states. In case users are unable to find vulnerabilities for an input state, LIMBO provides an option to make a synthetic

buffer overflow by calling a function that triggers a stack buffer overflow. The goal expressions can be setting a register (*e.g.*, %ebx = 0x0804a010), writing to a memory (*e.g.*, Memory[0x0804842f] = 0xB), reading from a memory to a register (*e.g.*, %eax = Memory[0x0804a018]) or executing commands (*e.g.*, system("/bin/sh")). Both BOPC and LIMBO use concolic execution and heuristics to make state transitions. However, the key difference between these two techniques is how the two techniques look for the goal state. BOPC divides an exploit goal into several smaller goals, confirms the reachability of the smaller goals, and combines the smaller goals to achieve the targeted goal. On the other hand, LIMBO considers the exploit goal as a single state and searches for the exploiting state through the concolic execution.

The difficulty of attack generation using data-oriented exploit generation tools depends on whether the tools support the end-to-end exploit generation. If the exploit generation tool does not generate end-to-end exploits, then attackers must manually set up the initial phase of an exploit, *e.g.*, finding vulnerabilities to control memory. Also, attackers must interpret the output of the automatic exploit generation tools. The interpretation of a tool's output can be straightforward. However, the setup of the initial phase can be challenging and varies between tools. For example, BOPC requires arbitrary memory read/write primitives and an entry point for the initial phase. Similarly, LIMBO requires control over a program state by triggering vulnerabilities (*e.g.*, triggering stack buffer overflow to control a stack frame). On the other hand, STEROIDS requires the gadget lookup and gadget stitching methodologies to be provided by attackers. STEROIDS also requires the attackers to trigger a memory corruption vulnerability. Thus, in most cases, the initial phase of automatic exploit generation tools requires the discovery and triggering of vulnerabilities to control and leak memory.

Address space layout randomization can exacerbate the setup of the initial phase. In the presence of coarse-grained ASLR [199], attackers require one or more vulnerabilities to control a program's flow as well as to leak memory. The memory leak is necessary for obtaining knowledge about the address space of a program. Moreover, a single memory leak is not sufficient in the presence of fine-grained ASLR (ASR [84], ASLP [108], CCR [111], Selfrando [47], etc.).

### 4.2.4 Block-Oriented Programming (BOP) Attack

Unlike DOP, BOP [105] constructs data-oriented exploits by chaining the basic blocks together instead of instructions. The core of BOP is the Block-Oriented Programming Compiler (BOPC). BOPC searches for the necessary basic blocks for an exploit. An exploit is written in a high-level C like language called SPloit Language (SPL) (Figure 4.2(a)). BOPC maps each SPL statement to a functional block and chains the functional blocks using a set of dispatcher blocks. The single- and double-bordered rectangles in Figure 4.2(b) represent dispatcher and functional blocks. A functional block executes the semantics of an SPL statement whereas a set of dispatcher blocks links between two functional blocks.

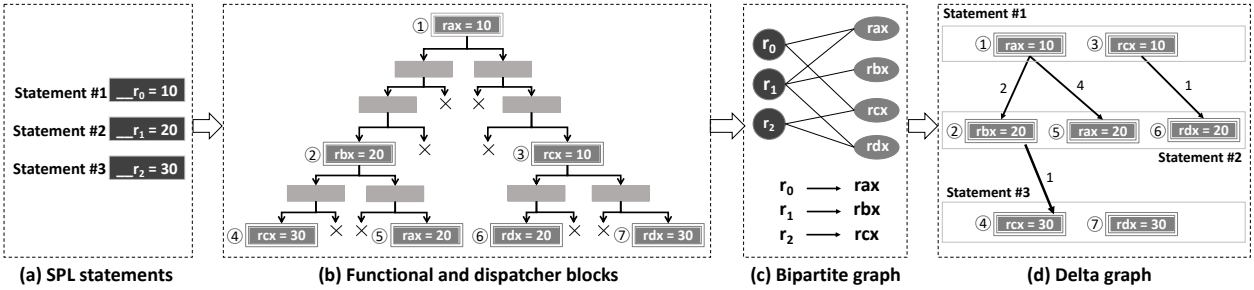



Figure 4.2: Four major components of a BOP Compiler. The double and single border boxes (□) indicate functional and dispatcher blocks. The number inside a circle (◯) represents the functional block number. The ✕ represents irrelevant basic blocks.

To search and select functional blocks for SPL statements, BOPC creates a bipartite graph by associating each SPL statement with a set of functional blocks that may potentially serve the SPL statement. Figure 4.2(c) shows a bipartite graph for the SPL statements in Figure 4.2(a) where functional blocks ① and ③ can serve SPL statement #1; functional blocks ②, ⑤, and ⑥ can serve SPL statement #2; and functional blocks ④ and ⑦ can serve statement #3. BOPC selects an association from many possible associations. Figure 4.2(c) shows one such association ($r_0 \rightarrow rax$, $r_1 \rightarrow rbx$, and $r_2 \rightarrow rcx$).

BOPC selects a set of dispatcher blocks by constructing a delta graph as an arbitrary selection of the dispatcher blocks may also clobber the SPL state. A delta graph is a multipartite graph that has functional blocks as nodes. An edge of the graph represents the basic blocks that are necessary

for moving from one functional block to the next one with the numbers of basic blocks as edge weights. Figure 4.2(d) shows a delta graph for the SPL statements in Figure 4.2(a). A recursive version of Dijkstra's [48] shortest path algorithm minimizes the set of basic blocks required for moving from one functional block and another. This minimization process produces a sub-graph of the delta graph indicated by bold edges in Figure 4.2(d).

Once the subgraph is created, BOPC selects a functional block and translates the basic blocks between the selected and the next functional blocks into constraints by leveraging concolic execution [171]. Once the last functional block is reached, BOPC checks for satisfying assignments for these constraints. If satisfying assignments are possible, BOPC produces a BOP gadget chain. BOPC does not produce an end-to-end exploit program. BOPC requires an entry point where the execution of the exploit payload should start and outputs a set of "what-where" memory writes to indicate the memory initialization and memory setup to execute the exploit payload.

### 4.2.5 Comparison Between DOP and BOP

Both DOP and BOP are data-oriented attacks. However, DOP is focused more on the generalization side of data-oriented attacks, whereas BOP is focused on the automation side of data-oriented attacks. Thus, BOP has a few key differences from DOP in terms of *i)* automatic exploit generation, *ii)* exploit writability and *iii)* granularity. In DOP, one must analyze and construct exploits manually, whereas one can write BOP exploits using SPL, an easily understandable high-level C-like language. In terms of granularity, DOP works at the gadget level whereas BOP works at the basic block level. Note that BOPC is not fully automated. BOPC's output is a set of address-value pairs for memory writes. An attacker requires to modify an address with the corresponding value from the address-value pairs to launch an attack.

The authors of DOP [101] also defined a mini-language called MINDOP to express DOP exploits, in which the attacker's payload can be specified. SPL specifies BOP exploits and allows the explicit access of virtual registers, library functions, and APIs to call OS functions. MINDOP uses a set

of virtual instructions and virtual register operands. Both MINDOP and SPL are Turing-Complete languages. That means both languages support memory read/write, assignment, arithmetic operations, logical operations, control-flow transitions (*e.g.*, jump), function call, and system API calls.

It is easier to construct a BOP-based exploit than a DOP-based exploit because the gadget stitching for DOP-based exploits may depend heavily on the target program. Since the DOP gadget space is a superset of the BOP gadget space (a BOP gadget is a DOP gadget within a single basic block), it is relatively easy to find the necessary DOP gadgets from a program. But the construction of an exploit through stitching the gadgets is a challenging manual effort. Both types of exploits require memory-write primitives to stitch gadgets. However, BOPC writes memory with respect to the stack pointer and base pointer whereas we observe that the ProFTPd DOP exploit requires addresses from the address space of the ProFTPd server. The key challenge for BOP-based exploits is to find entry points. When a program's execution reaches the entry point, the BOP exploit takes over and executes the BOP payload. But, to reach the entry point, an attacker must find a vulnerability that allows the attacker to control the program control flow. Attackers usually craft external inputs with the exploit payload to trigger the vulnerability and to set up the memory for executing the exploit payload.

Although BOPC is not designed to implement a complete end-to-end attack, we provide an example to illustrate how BOPC makes the construction of a BOP exploit easier than that of a DOP exploit. To assess the difficulty of constructing a BOP exploit, we constructed a BOP exploit using the following SPL program (Listing 4.3) to invoke the `execve()` system call with `'/bin/sh'` as the argument. To construct the exploit, we ran the BOPC tool in Nginx (version 1.3.9) web server. BOPC took around 30 minutes for the basic block abstraction process and around 5 minutes to find a solution for the exploit. We ran the BOPC tool in an Ubuntu 18.04 machine with 16 GB of memory and an 8-core CPU. We ran the Nginx web server and used GDB to avoid the complication of finding and triggering a vulnerability to divert Nginx's control flow and set up an entry point for the exploit. We manually crafted the entry point of the exploit with the entry point

of the Nginx program obtained through GDB. Now, when we ran the Nginx program, we observed the invocation of the `execve()` system call and execution of the `dash` program. However, it requires non-trivial manual efforts to stitch the gadgets for constructing a DOP exploit to achieve the same attack purpose.

```
1 void payload()    // execve('/bin/sh') payload
2 {
3       string prog = "/bin/sh\0";
4       int argv       = {&prog, 0x0};
5       __r0 = &prog;
6       __r1 = &argv;
7       __r2 = 0;
8       execve(__r0, __r1, __r2);
9 }
```

Listing 4.3: SPL payload for invoking execve() system call.

### 4.2.6 Data-oriented Attacks on Real-World Applications

The existence of single-step DDM attacks [35, 100] in programs is not new. However, the advanced data-oriented attacks are new and pose serious threats to real-world programs.

Jia *et al.* [106] utilized data-oriented attacks to bypass the same-origin policy (SOP) enforcement in the Chrome browser. By manipulating the values of in-memory flags related to SOP security policy checking (which requires an arbitrary read/write privilege), the SOP enforcement can be undermined in Chrome. Davi *et al.* [61] showed that a data-only attack on page tables can undermine the kernel CFI protection. By manipulating the memory permissions in kernel page entries, the attack makes kernel code pages writable and subsequently enables malicious code injection to kernel space.

Rogowski *et al.* [162] introduced a new technique, called memory cartography, that an adversary can use at runtime to reach security-critical data in process memory, and then modify or exfil-

trate the data at will. They demonstrated the feasibility of data-oriented exploits against modern browsers such as Internet Explorer and Chrome, where possible attacks range from cookie leakage to bypassing the SOP. Morton *et al.* [133] demonstrated the potential threat of data-oriented attacks against asynchronous web servers (*e.g.*, Nginx or Apache). By manipulating only a few bytes in memory, an attacker can re-configure a running asynchronous web server on the fly to degrade or disable services, steal sensitive information, and distribute arbitrary web content to clients. The attack consists of multiple steps (*i.e.*, a multi-step DDM). It starts with locating the security-critical configuration data structures of the server and exposing their low-level state by leveraging memory disclosure vulnerabilities. Then, an adversary constructs faux copies of security-critical data structures into memory by exploiting memory corruption vulnerabilities. By redirecting data pointers to faux structures, a running web server instance can be re-configured by the attacker without corrupting the control-flow integrity or configuration files on disk. However, in the end-to-end exploits, authors in [133] simulated the arbitrary write vulnerability in the recent version of Nginx, rather than exploiting a real-world vulnerability.

Table 4.1 summarizes these recent data-oriented attacks targeting real-world applications. Because existing CFI-based solutions are rendered ineffective under data-oriented exploits, such threats are particularly alarming. To construct a data-oriented exploit, attackers must have an in-depth knowledge of the vulnerable program's exact memory layout at runtime. In comparison to the DDM attack, a DOP attack requires non-trivial engineering efforts to chain gadgets for malicious effect.

## 4.3 Existing Defenses Against Data-Oriented Attacks

We describe a three-stage model for data-oriented attacks, a taxonomy of existing defenses, and a comparative analysis of existing defenses against data-oriented attacks in this section.

Table 4.1: Recent data-oriented attacks pose serious threats against real-world programs.

| Targeted Application and Year | Type | Assumption/Requirement | Capability/Attack Purpose |
|---|---|---|---|
| *Chrome* [106], 2016 | DDM | Identified security-critical variables, and arbitrary read/write capability | Bypass the same-origin policy |
| *Linux Page Table* [61], 2017 | DDM | Kernel code writable, and arbitrary read/write capability | Bypass the kernel CFI |
| *Internet Explorer, Chrome* [162], 2017 | DDM | Identified security-relevant variables, and arbitrary read/write capability | Information leakage, bypass the same origin policy, etc. |
| *Nginx* [133], 2018 | Multi-step DDM | Identified security-critical data structures, known unused portion of the data section, and arbitrary read/write capability | Disable or degrade services, information leakage, etc. |
| *Nginx* [73], 2015 | DDM | A stack vulnerability that allows an attacker to corrupt a data pointer | Leakage of safe code pointers stored in secret locations, *i.e.*, bypassing the CPI [115] protection. |
| *Apache HTTP Server & Glibc* [173], 2014 | DDM | A stack-based buffer overflow that allows attackers to corrupt data variables, data pointers, and code pointers | Derandomize code layout by learning how code is diversified without a memory disclosure vulnerability |
| *Nginx & Sudo & Httpdx & Orzhttpd & Null Httpd & Ghttpd & Sshd & Wu-ftpd* [100], 2015 | DDM (Flow-Stitch) | Identified source and target data flows, and arbitrary read/write capability | Automatic construction of DDM exploits by stitching disjoint data-flows via exploiting memory errors (information leakage or privilege escalation attacks) |
| *ProFTPd* [101], 2016 | DOP | Memory addresses of multiple involved data, identified gadgets/dispatchers, and arbitrary read/write capability | Private key leakage w/ ASLR |
| *ProFTPd, Nginx, Sudo* [105], 2018 | DOP (BOPC) | Attack payload written in SPloit Language (SPL), and arbitrary read/write primitive | Automatic construction of BOP gadget chain or exploit payload |

### 4.3.1 Three-stage Model for Launching Data-oriented Attacks

Figure 4.3 illustrates the abstract view of three stages in data-oriented attacks. To launch such attacks, it starts with triggering a memory error of a vulnerable program (*i.e.*, Stage S1), which empowers an attacker with control of the memory space, *e.g.*, read/write capability. In Stage S2, the targeted non-control-data is modified (through either DDM or DOP). In Stage S3, the manipulated data variable is used and takes effect to change the default program behavior. Note that S3 does not necessarily happen immediately after the data manipulation. The back edges pointing from S3→S1 and S2→S1 indicate that an attacker may need to corrupt non-control-data multiple times

to achieve the malicious goal.

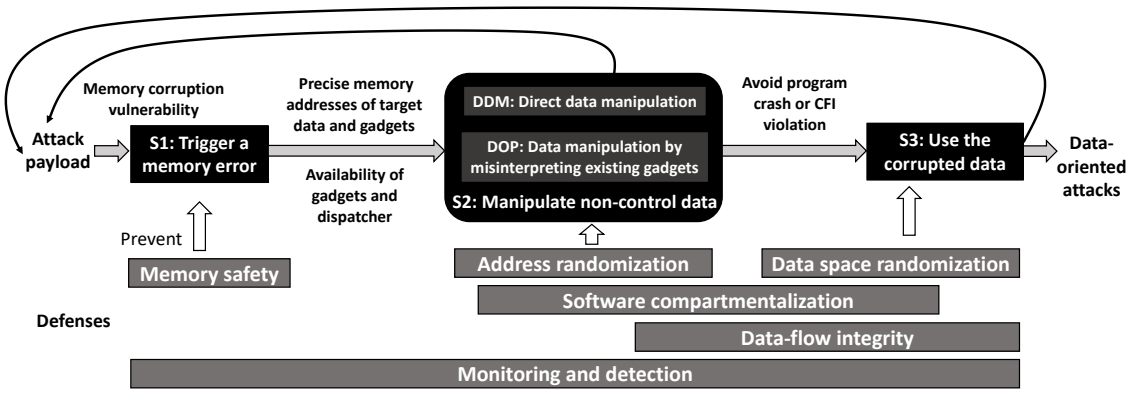



Figure 4.3: Stages in data-oriented attacks and mitigation in different stages

We discuss requirements in different stages (*i.e.*, the threat model) that are essential to launching a successful DOP attack. The first three requirements apply to DDM exploits.

1. The *presence of a memory corruption vulnerability* (such as a buffer or heap overflow) in the target program, which allows attackers to modify the content of the program's memory (*i.e.*, write capability). This assumption is reasonable since memory-unsafe languages (*e.g.*, C/C++) are still widely used today for their interoperability and speed.

2. Knowing the exact *location of target non-control data in memory*. Due to the wide deployment of exploit mitigation technologies such as DEP and ASLR, it is likely attackers need to first leverage memory disclosure vulnerabilities to circumvent the address space randomization [133]. In this case, an exfiltration channel to achieve information leakage is needed (*i.e.*, read capability), such as reading data from arbitrary addresses of the target program.

3. Knowing exactly the *impact of an exploit on the memory space* of the target program. For example, a continuous buffer overflow exploit may generate side effects that cause the program to crash. When launching a data-oriented exploit, attackers need to avoid any CFI violation and program crash.

4. *Availability of DOP gadgets* that are reachable using a memory corruption and triggerable using an exploit

5. *Stitchability of disjoint DOP gadgets*. A gadget dispatcher dispatches and executes the functional DOP gadgets. However, it is non-trivial to find gadget dispatchers in a program since they require loops with suitable gadgets and selectors controlled by a memory error.

Overall, constructing DOP exploits imposes some restrictions on the nature of the vulnerability, the context of the execution, and the defenses under deployment. Next, we discuss applicable defenses focusing on preventing these requirements from being satisfied at different points/stages. More generic memory corruption prevention mechanisms (in Stages S1 and S2) can be found in [117, 189, 196]. These defense techniques can prevent general types of memory corruption attacks, which apply for both control-flow attacks and data-oriented attacks. In addition to generic memory corruption prevention mechanisms, a number of detection and prevention techniques specially focusing on data-oriented attacks have been proposed in the literature. We then discuss these defensive mechanisms in Section 4.3.5.

### 4.3.2 S1 Defense – Preventing the Exploitation of Memory Errors

*Memory safety* enforcement is the first line of defense, which aims to prevent both spatial and temporal memory errors, such as buffer overflow, use-after-free, etc. It ensures the low-level integrity of a program's data structures and avoids invalid memory accesses. Memory-safe programming languages have built-in mechanisms to protect memory errors. In contrast, *memory-unsafe* languages such as C/C++ lack built-in memory safety guarantees, hence memory errors are prevalent in programs written in these languages. They allow direct access to memory using pointers, which is a common cause of memory corruption. Nevertheless, C and C++ are still widely used programming languages today [76]. Despite considerable prior research in retrofitting memory-unsafe programs with memory safety guarantees, memory-safety problems persist due to a trade-off between effectiveness and efficiency: approaches with low-overhead usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance-overhead or provide limited backward compatibility [183, 196]. The majority of existing memory safety solu-

tions can be generally classified into two categories: pointer safety (*i.e.*, pointer-based approaches focusing on pointer dereference operations, including pointer-based bounds checking and pointer integrity/authenticity) and object safety (*i.e.*, object-based approaches focusing on pointer arithmetic operations).

**Pointer-based bounds checking** Pointer safety is typically realized by associating a lower and upper bound with each data pointer, and adding a check at runtime that verifies that memory accesses via the pointer fall within those bounds. Numerous pointer safety mechanisms based on such *pointer bounds checks* have been proposed. *SoftBound* [134] and *HardBound* [66] perform pointer bounds checks against metadata stored in a shadow memory area. The bounds information for each pointer must be frequently retrieved from the shadow memory. SoftBound adds software checks to applications hardened with it, but breaks cache locality when retrieving pointer bounds. As a result, it leads to additional cache misses which hurt program performance. SoftBound incurs an average performance overhead of 67% in standard benchmarks. HardBound is a hardware-assisted scheme where the processor checks associated pointer bounds implicitly when a pointer is dereferenced. As the check is performed by hardware logic, the average performance overhead is reduced to $\sim$10%. Both schemes have a worst-case memory overhead of $\sim$200%.

Intel's Memory Protection Extensions (MPX) is an Instruction Set Architecture (ISA) extension for pointer safety introduced to Intel x86-64 processors in the late 2015 Skylake microarchitecture (MPX support was deprecated and removed from GCC and the Linux kernel [45]). MPX adds four new 128-bit registers for storing upper and lower pointer bounds and new instructions for managing the bounds registers and performing bounds checks on pointers. Bounds checks using the bounds registers are highly efficient. But since the number of bounds registers is limited, bounds information is also stored in tables with an index derived from the pointer address, similar to a two-level page table structure in x86. A 2GB intermediate table (bounds directory) is used as a mediator to the actual 4MB-sized bounds tables, which are allocated on-demand by the OS when bounds are created. The hardware performs a table walk of the bounds directory and bounds tables when bounds information is fetched to the registers. Oleksenko *et al.* [145] found that MPX

incurs an average performance overhead of 50% and a memory overhead of ∼90%, largely due to the complexity of storing and loading bounds metadata.

*Fat-pointer* schemes store the associated bounds metadata [113] together with pointers, *e.g.*, by increasing their length [137] or by borrowing unused bits from pointers [113]. Repurposing parts of a pointer to store validation data has the advantage of enabling fast retrieval of pointer metadata without a need for lookups from disjoint memory. But it changes the representation of pointers in memory in ways that break both binary and source code compatibility. Fat-pointers have primarily been deployed in clean-slate ISA designs [116], and memory-safe programming languages, *e.g.*, Cyclone [56] and Rust [200]. BIMA [116] is a hardware-assisted fat-pointer scheme for the SAFE secure computing platform [166]. BIMA limits the virtual addresses to 46 bits and restricts pointer alignment to powers of two. This frees 18 bits in 64-bit pointers for encoding bounds information. BIMA demonstrates that on a clean-slate ISA design, fat pointers can be realized without a performance penalty, and a 3% memory overhead due to segmentation caused by alignment restrictions on BIMA pointers. *Low-fat-pointers* [68, 69] are an alternative to fat pointers compatible with commodity 64-bit hardware architectures, such as x86-64. Low-fat-pointers require customized stack and heap allocators that restrict both stack frame and heap memory allocation sizes to a fixed finite set, and split the main program stack and heap into several sub-stacks and sub-heaps, one for each possible allocation size. Pointer accesses are then validated according to the allocation bounds associated with the corresponding sub-stack or sub-heap. The improved compatibility comes at the cost of accuracy, as low-fat-pointers accesses are only enforced at allocation bounds. On average, low-fat-pointers adds a performance penalty of 54% (16% for out-of-bounds writes) and memory overhead of 15% for stack data, and incurs a 56% performance (13% for out-of-bounds writes) and 11% memory overhead for heap data. Yong *et al.* [221] presented a security-enforcement tool for C programs preventing unchecked pointer dereferences. The proposed method uses static analysis to identify unsafe pointers in the program, as well as the memory locations that can be the legitimate targets of these pointers. To reduce the runtime overhead, only write instructions via unsafe pointers are instrumented to check for violations. An attack involving an attempt to write

to non-allocated storage, or to an inappropriate location on the stack (*e.g.*, the return address) will be detected.

Write-Integrity Testing (WIT) [6] uses points-to analysis to compute the control-flow graph and the set of objects that can be written by each instruction in a program. Then it generates code instrumented to enforce write integrity, which prevents instructions from modifying objects that are not in the set computed by the static analysis. To achieve a low runtime overhead, WIT only instruments writes without instrumenting reads, and thus is considered an S1 defense. It does not prevent out-of-bounds reads. The authors also developed several optimizations to reduce the space and time overhead in the implementation. WIT achieves a low average overhead of 7%, and the maximum overhead is 25% across a set of CPU intensive benchmarks. It is an approximation of the spatial memory safety (*e.g.*, preventing out-of-bound writes) in terms that the data write integrity typically maintains bounds based on static points-to analysis. Before each write dereference, it checks whether the location is within its valid points-to memory region. WIT mitigates use-after-free (UAF) vulnerabilities to a certain extent, because the attacker cannot use a dangling pointer to write to an object of a different equivalence class.

**Pointer Authenticity/Integrity**. *Pointer authenticity/integrity* aims to ensure the validity of pointers, *i.e.*, the value of a pointer (the address of the target object) is not arbitrarily controllable by an attacker, even in the presence of memory corruption vulnerabilities that may allow a manipulation over the pointer value. *PointGuard* [51] encrypts all pointers at runtime by XORing them against a key generated at program initialization. The encryption on each pointer must be reversed before dereferencing a pointer. PointGuard incurs a small to medium overhead (0%$\sim$20%), but is vulnerable to information disclosure, *e.g.*, if an attacker learns the key or the XORed ciphertext of a pointer to a known address. *Code-Pointer Integrity* (CPI) [115, 210] provides control-flow hijacking protection rather than complete memory safety. Therefore, it incurs a very low performance overhead with around 1.9% (C program) or 8.4% (C/C++ program) slowdown. Kuznetsov *et al.* [115, 210] also introduced a relaxation of CPI with better performance properties, called code-pointer separation (CPS), to achieve better security-to-overhead trade-off. However, this solution only protects

code pointers with non-control data unchecked.

*Pointer Authentication* (PA) [158] is a hardware pointer authenticity primitive introduced in the ARMv8.3-A processor architecture to protect programs from exploiting memory vulnerabilities. PA introduces a set of new instructions for calculating and verifying a *Pointer Authentication Code* (PAC) for pointers. The use of an unauthenticated pointer would cause a memory translation fault. Each PAC is generated using a key from a set of five different keys and a modifier. The kernel generates the five keys for each process and stores them in internal CPU registers which are not accessible from userspace code. These keys remain the same throughout the process lifetime. Out of the five keys, two are used for generating PACs for code pointers, two for data pointers, and one for general purpose uses. The modifier usually captures the contexts of pointer declarations and accesses. To store PACs, PA uses the unused bits in the virtual address of 64-bit address space. In a 64-bit Linux kernel, PA uses 24 bits for the PACs, but the size can vary based on memory scheme and address tag usages. However, PA has a few concerns regarding the PAC generation. Since PAC generation keys stay the same for the lifecycle of a process, and modifiers may have repeatability, attackers may reuse previously generated PAC and pointer pair at a later stage to replace another PAC and pointer that uses the same modifier [121]. If the modifier does not uniquely capture the context, it might repeat in different contexts and allow such reuse attacks. For example, in return address signing using the stack pointer (SP) values as modifiers, a return address authenticated for one function can be used for another function if the SP values in both functions are the same. To address these concerns, the Pointer Authentication Run-Time Safety (PARTS [121]) technique augments the PA-based defense approach and compartmentalizes the PAC generations for different pointers in different contexts. The key idea of the PARTS approach is to utilize a pointer's type as a modifier. The type potentially captures the context in which the pointer is created and dereferenced. PARTS provides a proof-of-concept implementation based on LLVM and incurs an overhead of less than 20%.

Pointer-based approaches generally suffer from poor scalability in terms of increased execution time and memory consumption as the number of protected pointers increases. They also require a

comprehensive understanding of a program's memory layout at individual pointer granularity over time in order to differentiate between benign (within bounds) memory accesses from malicious (out-of-bounds) memory accesses. Another concern is the compatibility problem with unprotected modules, which could modify or dereference signed pointers and result in false alarms.

**Object-Based Approaches**. Instead of enforcing bounds checking with pointers, object-based approaches detect out-of-bounds memory accesses to objects. It solves the compatibility issues caused by pointer-based approaches. *AddressSanitizer* (ASan) [176] is a memory error detector for Linux available in GCC and Clang/LLVM. It can detect out-of-bounds memory accesses to global, stack, and heap objects. In addition, it can detect a number of *temporal memory errors*, such as use-after-free and double free conditions. ASan tracks objects stored in application memory by storing metadata on each object in a disjoint *shadow memory* area that occupies a fraction of the application's virtual memory space. The shadow memory records which memory regions in the application memory are allocated and used, and therefore safe to access. However, these memory safety guarantees do not preclude erroneous memory accesses in which a corrupted pointer dereferences a valid, but unintended memory object. In addition, ASan places blocks of "poisoned" memory between adjacent objects in the stack, heap and global storage (*i.e.*, blacklisting unsafe memory regions). Different from approaches that whitelist safe memory regions, poisoned memory is marked as invalid in the application's shadow memory, and acts as "red-zones", which, if accessed, indicates a contiguous overflow, *e.g.*, beyond the array boundary. However, memory errors that enable non-contiguous accesses or accesses with a larger step distance than the size of the red-zone can violate spatial safety without setting off the tripwire. Hardware-assisted AddressSanitizer (HWASAN) [94] is a tool similar to AddressSanitizer, but based on partial hardware assistance. It relies on address tagging support which is only available on ARM's 64-bit architecture (AArch64).

### 4.3.3 S2 Defense – Providing a Barrier to Access to Data or Guess Memory Layout

The purpose of S2 defenses is to mitigate the consequences of attacks in the presence of memory vulnerabilities, including *software compartmentalization* [70, 129, 212] and *address randomization (diversification)* [15, 199] techniques. They serve as the second line of defense, which creates a barrier for attackers trying to access target data or infer memory layout.

**Software Compartmentalization**. Software compartmentalization isolates software components into distinct protection domains in order to limit the utility of existing memory errors (*i.e.*, when the memory error and data to be manipulated exist in different protection domains), but also limit the abilities of a compromised software component. For example, *Software Fault Isolation* (SFI) [212] compartmentalizes software in a single address space by sandboxing distrusted modules into separate fault domains, which are arranged to occupy a distinct portion of the program's address space. SFI-enforcement ensures code in the fault domain is unable to directly access memory or jump to code outside the reserved portion of address space. It can only interact with code outside its domain through well-defined call interfaces.

*XFI* [70] is a SFI variant for Microsoft Windows for isolating shared libraries within an application, or drivers within the kernel. However, XFI does not protect against *confused deputy attacks*, where a distrusted module abuses an over-permissive kernel routine that the module is allowed to invoke. *LXFI* [129] extends SFI to Linux kernel modules, which exhibits a more complex interface, *e.g.*, callbacks invoked by the kernel make manual interposition more difficult. LXFI also enables compartmentalization between different instances of a single module, *e.g.*, a kernel driver which may instantiate server module principals, such as block devices or sockets. *CHERI* [218] is a hardware-assisted capability model for the 64-bit MIPS ISA that can support different protection models, such as pointer safety and software compartmentalization [202, 214].

**Address Randomization**. *Address randomization* aims to hide attack targets by randomizing the

location of program segments [117], layout of the code (instruction set) [52], or layout of data [15] so that attackers are unable to predict the target data or code location from an address space. In particular, data space randomization [14, 15, 84] aims to randomize the locations of data stored in program memory at runtime to make the locations unpredictable, and thus reducing the possibility that attackers can leak security-critical memory addresses or manipulate the content of targeted data. ASLR [199], also known as coarse-grained ASLR, randomly relocates only the base addresses of the stack, heap, code segments, and shared libraries on each execution of a program. However, the internal layout of a segment or module remains unchanged. Also, to relocate the code and data segments of the main executable of a program, it is necessary to run the program as position-independent code (*i.e.*, with the PIE feature enabled). Fine-grained ASLR techniques relocate the internal layout of a segment or module up to different granularities such as function-level [47, 84, 108], basic block-level [38, 111, 213], instruction-level [97], and machine register-level [52, 98]. Attackers can still figure out the fine-grained address space layout of a program within a few seconds [5] using advanced attack techniques [185].

Though strong randomization can stop memory corruption attacks with a high probability, the protection is confined to all data/addresses that are randomized/encrypted. In practice, to avoid a significant performance degradation, not all data/addresses are protected by randomization defenses [196]. On the other hand, information leaks can undermine randomization techniques [185]. ASLR-Guard [126] can add an extra layer of protection for randomization techniques suffering from information leaks. Besides, ASLR-Guard [126] can render the code pointer leak through a data pointer useless by separating the data and code using a custom linker that generates new relocation information to fix the offset between the data and code. In addition, data/address encryption based solutions are not binary compatible (*i.e.*, protected binaries are incompatible with unmodified libraries) [196].

### 4.3.4 S3 Defense – Preventing/Detecting Use of Corrupted Data

Data-Flow Integrity (DFI) [29] mitigates data corruption before the manipulation takes effect. Before each read instruction, DFI ensures that a variable can only be written by a legitimate write instruction which can be derived by reaching definitions analysis. For each read instruction for reading a value, it statically computes the set of write instructions that may write the value, and assigns an identifier to each definition. DFI enforces a simple safety property, *i.e.*, whenever a value is read (used), the definition identifier of the instruction that wrote the value should be in the set of reaching definitions for the read. DFI is different from the data write integrity checking in S1 defense (such as WIT [6] ) in that it enforces data integrity for memory reads. Thus, the enforcement of DFI happens in Stage 3. WIT prevents data manipulation by protecting against invalid/unintentional memory writes, but with reads left unchecked. In addition, since DFI could potentially limit the DOP gadget availability, it is active at both S2 and S3 in our three-stage model.

DFI enforcement can prevent both control-data (*e.g.*, overwriting the return address) and non-control-data attacks. However, DFI usually overestimates the set of valid write instructions since the set is statically determined without runtime information. Moreover, Software-based DFI incurs a high performance overhead [101] due to the frequent read instruction checking. Intra-procedural DFI incurs 44% and inter-procedural DFI incurs 103% runtime performance overhead, respectively, and approximately 50% space overhead for instrumentation [29]. Hardware-based DFI, *e.g.*, HDFI [188], is efficient, but limited by the number of simultaneous protection domains it can support. Carlini *et al.* [27] have recently revealed fundamental limits on the effectiveness of CFI, and presented the Control-Flow Bending (CFB) which allows an attacker to "bend" the control-flow of a program but adheres to CFI's security policies, *i.e.*, modifying indirect branch targets that are valid based on a CFI policy. Depending on the granularity of compartmentalization and the boundaries of the security domain, software compartmentalization can also function as a defense in S3. It can prevent the use of corrupted data. For example, when a corrupted pointer is referencing memory in another protection domain, it thwarts the dereference operation. Besides,

some techniques [62, 197] can prevent the execution of code pointed by corrupted data or memory. Heisenbyte [197] is such a technique that utilizes "destructive code read". The "destructive code read" allows the execution of code as part of the normal flow of a program and restricts the execution of the same code when used in a dynamic code-reuse attack. Another technique called Isomeron [62] tackles the usage of existing code pointed by corrupted data or memory by randomizing the execution paths.

Data Space Randomization (DSR) [15] encrypts data (*i.e.*, randomizes the representation of data objects with a unique random mask) stored in memory, rather than randomizing the location of data objects. When a variable is used, its value will be first derandomized by using the unique random mask associated with the variable. By using different masks for different variables, DSR ensures that even if the attacker manages to overwrite a target variable, the attacker is only able to write a random value into it rather than the intended value. Instead of using a different mask for each data object, Data Randomization [26] groups data objects into different classes, assigns a random mask for each class, and generates instruments code to XOR data read from or written to memory with the corresponding mask. As a result, accesses that violate the read or write integrity have unpredictable results. Another recent DSR technique called CoDaRR [159] continuously re-randomizes the masks used in variables with load and store instructions while being transparent to program execution. The dynamic nature of CoDaRR prevents disclosure attacks. DSR [15, 159] and Data Randomization [26] are effective against non-control data attacks by preventing attackers from using the corrupted data in the memory space (although they can overwrite target data variables). And thus they actually take effects at Stage 3. Nevertheless, masking/unmasking every memory access inevitably incurs nontrivial runtime overheads, which hinders their practical deployments.

Szekeres *et al.* [196] highlights in their paper that real-world software exploits are still possible because memory vulnerabilities continue to grow and currently deployed defenses are being bypassed. Thus, program anomaly detection [42, 182, 220, 223] may complement the aforementioned mitigation techniques, and may serve as the last line of defense against data-oriented attacks. As

shown in Figure 4.3, passive monitoring based program anomaly detection has the potential to detect anomalous program behaviors exhibited in all the three stages.

### 4.3.5 Defense Mechanisms Especially Against Data-oriented Attacks

Besides the generic memory corruption prevention mechanisms which can be applied to defeat data-oriented attacks, here we discuss detection and prevention techniques explicitly focusing on data-oriented attacks.

*YARRA* [168] is a C language extension that validates a pointer's type for *critical data types* annotated by developers. It guarantees that critical data types are only written through pointers with the given static type. YARRA is suitable for hardening access to isolated pieces of critical data, such as cryptographic keys stored in program memory at runtime. However, when applied for the whole program protection, it incurs performance overhead around 400%–600%. Besides, YARRA relies on the programmers' manual annotations, which is undesirable for complicated programs.

*HardScope* [142] is a hardware-assisted variable scope enforcement approach to mitigate data-oriented attacks. It performs intra-program memory isolation based on C language variable visibility rules derived during program compilation. On each memory access (*i.e.*, load/store), HardScope enforces that the memory address requested is in the accessible memory areas. Nyman *et al.* [142] demonstrated the effectiveness of HardScope for the RISC-V architecture, by introducing a set of seven new instructions. HardScope instruments instructions at compile-time and enforces memory access constraints at runtime. HardScope's performance overhead is reasonable with 3.2% in embedded benchmarks. Although HardScope significantly reduces the usefulness of DOP gadgets and thwarts Hu *et al.* [101]'s example attacks, HardScope cannot guarantee the absence of DOP gadgets in arbitrary programs.

*PrivWatcher* [34] is a framework for monitoring and protecting the integrity of process credentials (*i.e.*, *task_struct* that describes the privileges of a process in the Linux kernel) against non-control

data attacks. PrivWatcher provides non-bypassable integrity assurances by relocating process credentials into a safe region, code instrumentation and runtime data integrity verification. It incurs more than 94% overhead for applications that involve installing new *task_struct* structures to processes.

*Hardware-Assisted Data-flow Isolation* (HDFI) [188] extends the RISC-V architecture to provide an instruction-level isolation by tagging each machine word in memory (also known as the tag-based memory protection). The one-bit tag of a memory unit in HDFI is defined by the last instruction that writes to this memory location. At each memory read instruction, HDFI checks if the tag matches the expected value. However, unlike software-enforced DFI, HDFI only supports two simultaneous protection domains.

*PT-Rand* [61] aims to protect a data-oriented attack against kernel page tables to bypass CFI-based kernel hardening techniques. To mitigate the manipulation of page tables, PT-Rand randomizes the location of the page tables. PT-Rand incurs a low overhead of 0.22% for common benchmarks on Debian. However, it is still possible attackers undermine these schemes if the secret information (*e.g.*, randomization secret) is leaked or inferred [142].

Both C-FLAT [4] and eFSA [41] identify non-control-data attacks through their effects on the control-flow of a program in embedded systems, such as Internet of Things (IoT) or Cyber-Physical Systems (CPS). C-FLAT [4] enables remote attestation of an application's control-flow path, which allows a verifier to detect control-flow deviations launched via code injection, code-reuse, and certain non-control-data attacks. Since C-FLAT computes an aggregated authenticator of the program's control flow, including branches and function returns, it can detect non-control-data attacks that affect a program's sequence of executed instructions or the number of loop iterations. eFSA [41] is an event-aware finite-state automaton model for detecting non-control-data attacks against programs in CPS. It takes advantage of the event-driven nature of CPS control programs and incorporates event checking in anomaly detection. It detects non-control-data attacks if a specific physical event is missing along with the corresponding event dependent executed instructions.

*CVI* (Critical Variable Integrity) [194] verifies define-use consistency of critical variables for embedded devices. The define-use consistency enforces that the value of a variable cannot change between two adjacent define- and use-sites. After identifying critical variables (either automatically identified or manually annotated), the compiler inserts instrumentation at all the define- and use-sites for these critical variables, to collect values at runtime and send them to an external measurement engine. CVI checking compares the current value of a variable at every use-site and the recorded value at the last legitimate define-site. However, like DFI [29], CVI is based on compile-time instrumentation and frequent runtime checking, which incurs a high overhead for the complete protection.

*Hardware-based detector I.* This detector [201] utilizes Hardware Performance Counters (HPCs) to collect hardware events related to instructions retired, cache-misses suffered, and branches mispredicted for detecting data-only exploits. The authors collect 12 hardware events during the normal and abnormal executions of OpenSSL and train a multi-class support vector machine model to classify normal and abnormal behaviors. Two fundamental limitations of this HPC-based detector is that *i*) the co-executing programs significantly affect HPC-based events and *ii*) HPC-based events are dependent on instruction set architectures (ISAs).

*Hardware-based detector II.* This detector [123] utilizes the HPC events as short time series by monitoring various execution regions of a vulnerable program. Especially, the samples include the region before, during, and after executing a vulnerable region of a program. This time series approach enables more fine-grained detection than using only hardware events. Classification algorithms such as the Stacked Denoising Autoencoder and Echo State Network classifiers are around 98% accurate for detecting data-oriented exploits.

In summary, HDFI [188], PrivWatcher [34], PT-Rand [61], and CVI [194] protect specifically non-control data. HardScope [142] can protect against all DOP attacks that violate variable visibility rules at runtime. However, its main drawback is that the new hardware extensions [188] set a high

bar for deployment. On the other hand, the two general approaches DFI [29] and YARRA [168] incur a high performance-overhead at runtime. In general, the HPC-based hardware events are significantly affected by co-existing programs. Thus, the filtration of the hardware events that are produced by co-existing programs is a critical step for obtaining accuracy and reliability by HPC-based detectors. The time-series approach used by the hardware-based detector II [123] makes the detector less independent of the underlying applications. Different from the above works focusing on traditional programs, C-FLAT [4] and eFSA [41] target at detecting non-control-data attacks in embedded systems by monitoring their side-effects on control-flow behavior.

### 4.3.6 Anti-specification Database for Detecting Data-Oriented Attacks.

Anti-specifications – a specification-based technique that describes the violation of legal data flows–can aid the detection of data-oriented attacks. Specifications are what a program should do, whereas anti-specifications [207] are what a program should not do. Anti-specifications aim to capture events that may be part of an exploit. For example, attackers trigger most exploits using unguarded external inputs. Anti-specifications provide a characterization of the impact of a program's use of unguarded external input on the downstream data-flow of the program. The impact analysis of a program's external input is a key step for preventing the gateways of data-oriented exploits. Besides, anti-specifications can also capture input-dependent predicates used in loops or conditions. These controllable predicates can alter program behavior. Other types of anti-specifications may capture data leaks [23] or data pointer corruption. Thus, anti-specifications can help improve the detection of data-oriented exploits by identifying the gateways and generic components used in those exploits.

One key benefit of anti-specifications is the construction of an anti-specification database. This database is a collection of anti-specifications in a specific format, which helps screen programs against the anti-specifications stored in the database. It could also be used to generate new anti-specifications by merging two or more anti-specifications. The key difference of this anti-

specification-based approach from a signature-based approach is that anti-specifications are not specific to an exploit. The approach attempts to produce anti-specification by breaking an exploit into its modular events or components and extracting anti-specifications from them. Even though the same or similar exploits vary from one program to another, we may observe many common components (*e.g.*, exploit entry, gadget lookup, gadget stitching, and triggering) when unfolding the exploits. As a result, anti-specifications built from such common components can be more generally applied across multiple programs. We manually constructed one such a database that is available at `https://github.com/salmanyam/antispec`.

# Chapter 5

# Data Object/Pointer Prioritization

## 5.1 Introduction

With the advances toward practical code pointer protection countermeasures [12, 49, 103, 104, 115, 126, 130, 158] and practical Control-Flow Integrity (CFI) [24, 83, 124, 130, 132, 152, 222, 224], we anticipate a shift towards the use of data object/pointer-manipulation as the attack vector as the manipulation works in the presence of code pointer protection and CFI countermeasures. This is why in recent years, we observed a momentum in data-oriented attacks (also known as non-control attacks) [100, 101, 106, 133, 162, 168, 196, 219] even though data-oriented attacks were introduced more than a decade ago [35].

Data object/pointer manipulation has become an appealing attack technique for data-oriented attacks. Data-oriented attacks confirm CFI and achieve malicious goals by changing program behavior. Ideally, DOAs [13, 35, 100, 106, 219] can modify all kinds of data objects to change program behavior to leak sensitive information [23] or perform privilege escalations [57]. However, data pointer overwrite is preferred because it allows attackers to corrupt data pointers to point to arbitrary and unintended locations [49]. For example, data pointer manipulation can leak critical information about an application's address space layout [73, 173]. Data-Oriented Programming (DOP) [101] requires the address of some con-control data pointers to accomplish DOP-based attacks. Chen *et al.* [35] corrupts a data pointer in the ghttpd HTTP server through a stack buffer overflow to bypass security checks of input strings. Besides, heap-based exploitations such as the *House of Spirit* attack [180] on Glibc also manipulate a data pointer returned by malloc().

To stop attackers from manipulating data objects/pointers (and data variables in general), researchers have proposed both software and hardware-based countermeasures. Software-based countermeasures such as Data-Flow-Integrity (DFI) [29], Data Space Randomization (DSR) [15, 26, 159], and memory tagging [134, 135]. DFI can block data-oriented attacks, but it usually overestimates the set of valid write instructions due to its static analysis. This overestimation incurs high overhead (e.g., intra-procedural DFI about 44% and inter-procedural DFI around 103% [29]). DSR encrypts data by changing representation of every variable [15] and a group of variables [26] by adding masks for a single round or continuously changing the masks [159]. Though the effectiveness of DSR, masking/unmasking every memory access inevitably incurs nontrivial runtime overheads. Software-based memory tagging countermeasures also cost a significant amount of overhead, from 48-116% [134, 135].

On the other hand, hardware-based countermeasures (e.g., HDFI [188], Intel's CET [103], ARM Pointer Authentication (PA) [158], and MPX [104]) is efficient, but in general, limited for one or a few platforms. Furthermore, the overhead is non-negligible. For example, ARM pointer authentication [158] and Intel's MPX [104] cost on average around 19.5% [121] and 50% overhead, respectively, for protecting data pointers. The 19.5% overhead for a hardware-based technique is still critical for performance-critical applications. Due to a huge number of data objects/pointers in an application compared to code pointers, one source of the overhead for protecting data objects/pointers is protecting many data objects/pointers that do not need protection as those objects/pointers do not lead to vulnerability. One way to reduce the performance overhead is to figure out the sensitive (*i.e.*, vulnerable) data objects/pointers and prioritize them for protection.

The idea of protecting sensitive or critical data is not new. Palit et al. designed a compiler-level defense that protects critical data [148, 149]. However, they manually annotate the sensitive data. Similarly, FlowStitch [100] performed the automation of data-oriented attacks using predefined critical data. Our work complements this work by identifying and prioritizing the sensitive data automatically. A few automated techniques [106, 133] also determined the critical data. For example, Jia et al. [106] determined the decision-making data by recording the execution of two traces

with normal execution and violated execution, and observing the data that get modified and change executions. Access-driven trace data [133] are also useful to determine and understand the critical data and their structures. However, these works are not scalable as we need huge and relevant execution and access traces. On the other hand, Pathfinder [162] can automatically navigate to sensitive data from a leaked data pointer. However, it does not indicate how to determine or label sensitive data.

The goal of this work is to develop a data object/pointer prioritization framework that is both i) generic and ii) adaptable. The generic nature of the framework ensures that it does not rely on underlying operating systems, platforms, or programming languages. The adaptability feature makes the DPP framework adaptable with minimum changes so that it can be used with other defenses (e.g., ARM PA [158], Intel MPX [104], AddressSantizer [175], memory tagging [134, 135], etc.).

Our prioritization framework uses rule-based heuristics to detect sensitive data objects/pointers. We analyze existing data-oriented exploits [100, 101], vulnerabilities (CVE-2001-0820, CVE-2006-5815, CVE-2006-5815, CVE-2017-9430, CVE-2018-6151, CVE-2018-10111, CVE-2021-23017, etc.), and exploit description [1, 35, 73, 100, 173] to extract seven generalized rule-based heuristics in four categories. We apply the heuristics on top of a tainted data/value flow graph of a program to detect/prioritize data objects/pointers. To construct the data flow graph, we utilize the interprocedural Static Value Flow (SVF) analysis [193]. We identify the taint sources and propagate the tainted data through the SVF graph. The seven rules detect vulnerable tainted data (i.e., taint sinks). To manipulate any data in a program, attackers must use the input channels/streams (e.g., network, file system, or keyboard) to pass their crafted commands or packets. Thus, we taint all data from input channels and propagate the tainted value throughout the SVF graph.

We implemented the taint analysis and rule-based heuristics as LLVM analysis passes in LLVM 12 [1]. To perform the pointer analysis and data-flow construction, we utilize the latest version of

[1] https://releases.llvm.org/12.0.0/docs/ReleaseNotes.html

SVF [193] compatible with LLVM 12. For performance evaluation, we have modified the Address Sanitizer tool [175] to add an option to instrument data objects/pointers prioritized by our rules. We have also implemented one instrumentation pass to instrument LLVM IR to include ARM pointer authentication instructions.

A key requirement to evaluate the effectiveness of our prioritization technique is ground truth vulnerable data objects/pointers. To the best of our knowledge, there exists no such dataset. We manually construct the ground truths vulnerable data objects by identifying 23 vulnerable data objects/pointers from 18 programs including 5 real-world applications. We used eight real-world applications and one benchmark program to evaluate the performance of our prioritization technique utilizing AddressSantizer [175] and ARM PA [158].

Our key contributions of this work are as follows.

- We proposed a prioritization framework that is both *i)* generic and *ii)* adaptable. The generic nature of the framework ensures that it does not rely on underlying operating systems, platforms, or programming languages. The adaptability feature makes the framework adaptable with minimum changes so that it can be used with other defenses such as ARM PA [158], MPX [104], Softbound [134], AddressSanitizer [175], and many more. One key use case of the framework is to tune the performance vs. security in an application.

- We extracted seven vulnerability- and exploit-driven rule-based heuristics to prioritize data objects/pointers that are sensitive and could potentially lead to vulnerabilities.

- We implemented our framework using eight analysis passes (one for taint analysis and seven for data object/pointer identification using our rules) on top of LLVM 12. We also implemented one instrumentation pass for instrumenting our prioritized pointers using ARM PA [158] and modified AddressSanitizer [175] tool to support instrumenting only the prioritized data objects/pointers in addition to AddressSanitizer's default behavior.

- To evaluate the prioritization framework, we constructed ground truths by manually ana-

lyzing vulnerable programs considering local/global data objects/pointers, inter-functional analysis, and corner cases. We constructed 33 ground truths data objects from 18 programs including real-world server applications and 10 test cases from SAR dataset.

- We found that as low as only 3% of total data objects are needed to protect for real-world applications. We achieved a 42% performance overhead reduction compared to AddressSanitizer while protecting 100% of the prioritized data objects. We can reduce the instrumentation size by around 62% and the number of pointers in Load/Store IR instructions (for ARM PA) by 56% and 96%, respectively, without compromising security.

## 5.2 Background and Threat Model

### 5.2.1 Pointer Manipulation

Arbitrary memory read/write capability is the gateway of control-oriented [18, 28, 169, 185] or data-oriented [35, 100, 101, 105, 172] attacks. Due to the prevalence of memory-corruption vulnerabilities (*e.g.*, user-after-free, type confusion, etc.) in C/C++ code, attackers demonstrated their capability to read/write arbitrary memory. The capability to manipulate arbitrary pointers is often targeted as it allows attackers to corrupt the pointers to point to arbitrary locations as their interests. Attackers utilize two kinds of pointers: *i)* code pointers and *ii)* data pointers.

*Code pointer* leaks serve two purposes for attackers: i) one kind of code pointer enables attackers to infer a program's layout, and ii) another kind of code pointer enables it to subvert a program's control flow. The first type of code pointers can help attackers to leak memory contents [9, 82, 96, 190, 192, 196] or help repetitively disclose information about a program's address space (*e.g.*, JIT-ROP's dynamic code harvest technique [185]). Attackers use the second type of code pointers to divert a program's flow to the attacker's chosen location (*e.g.*, modifying a function pointer pointing to shell-code). These two kinds of code pointers include function pointers, return addresses, data

objects (that point to composite data structures that contain function pointers), and virtual table pointers. Code pointers are practically feasible to protect within an insignificant overhead, *e.g.*, less than 1% overhead for C and 9% for C++ [115, 121].

*Data pointers* as attack vectors have gained attackers' interests with the advancement in protecting code pointers in relative low overhead. DOP [101] used the address of some con-control data pointers to accomplish DOP-based attacks. Chen *et al.* [35] corrupted a data pointer in the ghttpd HTTP server through a stack buffer overflow to bypass security checks of input strings. COOP [169] utilized a C++ object to hijack the *virtual table pointer* of a C++ object and constructed an exploit using the virtual functions as gadgets. BOP [105] used arbitrary memory read/write primitives and LIMBO [172] used to control over a program state by triggering vulnerabilities (*e.g.*, triggering stack buffer overflow to control a stack frame). Besides, heap-based exploitations such as the *House of Spirit* attack [180] on Glibc also manipulate a data pointer returned by malloc().

Attackers leak code/data pointers utilizing memory disclosure vulnerabilities through buffer overread [192], heap spray [177], and heap feng shui [190]. Attackers also use software side-channels such as guessing [18, 179], pointer probing [73], and timing side channel attacks [173]. Hardware side-channels (e.g., Load-evict-load [102], exploiting page walk in the page table hierarchy of Memory Management Unit [90], and exploiting the prefetch side-channel [91]) have become also popular in recent years for leaking code pointers.

Due to the widespread deployment of ASLR, attackers exploited data pointer manipulation to infer knowledge about the address space layout of a process to bypass ASLR defenses. Data pointer manipulation can cause some events that result in software side-channels—typically timing side-channels—that can leak information about the address space. To infer information about an application's address space, attackers analyze the output and execution time of a portion of code. They then correlate the output and timing information by running the same portion of code locally [73, 173]. For example, an attacker may overwrite a data pointer (`ptr` in Listing 5.1) to point to some byte sequences and infer knowledge about the byte sequences by observing the

output, *i.e.*, the variable `result`. By pointing the data pointer to different locations of address space and observing the output behavior, an attacker can distinguish the mapped and unmapped code pages.

```
1 struct mystruct {
2   int value;
3 };
4 void vuln_function()
5 {
6   char buf[64];
7   int result=0, length, input;
8   struct mystruct * ptr;
9   recv(socket, buf, input);
10  ptr->value = strlen(buf);
11  while (result < ptr->value) result++;
12  send(socket, &result, length);
13 }
```

Listing 5.1: Data pointer manipulation to infer knowledge about address space layout. This figure is adopted from [173].

All these exploits indicate that data pointers are as equally important attack vectors as code pointers. Unfortunately, software-based memory safety protection incurs a significant amount of overhead, ranging from 48% to 116% [68, 69, 134, 135, 136]. Data-Flow Integrity also protects memory safety issues but incurs overhead ranging from 44% to 103% [29, 187]. On the other hand, PointGuard [49] has relatively low overhead (up to 20%) for protecting pointers, but PointGuard's memory-related assumption that attackers cannot read arbitrary memory is no longer practical. Thus, software-based pointer protection, in general, is not practical due to a high runtime overhead.

Hardware-based solutions [66, 103, 104, 158, 188] can reduce the overhead significantly. However, the overhead is still non-negligible. This is due to the abundance of data pointers compared to the code pointers. This abundance of data pointers makes it difficult to build a CPI-like solution for protecting the integrity of data pointers. Thus, data pointer prioritization is extremely important and necessary to protect sensitive data objects/pointers. It is important to mention that over-approximation in prioritizing data objects/pointers is acceptable as long as the approximation

includes all sensitive data pointers and the overhead is in an acceptable range.

### 5.2.2 Memory Safety Defenses

*Memory-unsafe* languages such as C/C++ lack built-in memory safety guarantees, hence memory errors are prevalent in programs written in these languages. Nevertheless, C and C++ are still widely used programming languages today [76]. Despite considerable prior research in retrofitting memory-unsafe programs with memory safety guarantees, memory-safety problems persist due to a trade-off between effectiveness and efficiency: approaches with low-overhead usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance-overhead or provide limited backward compatibility [183, 196]. *SoftBound* [134] and *Hard-Bound* [66] perform data pointer safety by associating a lower and upper bound with each data pointer, and verifying the bound against metadata stored in shadow memory at runtime for C programs. *SoftBound* incurs an average performance overhead of 67% due to software-based bound check while *HardBound* performs the check using hardware logic that lowers the overhead to 9% on average. *Fat-pointer* schemes store the associated bounds metadata [113] together with pointers, *e.g.*, by increasing their length [137] or by borrowing unused bits from pointers [113]. Re-purposing parts of a pointer to store validation data has the advantage of enabling fast retrieval of pointer metadata without a need for lookups from disjoint memory. But it changes the representation of pointers in memory in ways that break both binary and source code compatibility. Fat-pointers have primarily been deployed in clean-slate ISA designs [116], and memory-safe programming languages, *e.g.*, Cyclone [56] and Rust [200]. BIMA [116] is a hardware-assisted fat-pointer scheme for the SAFE secure computing platform [166]. BIMA limits the virtual addresses to 46 bits and restricts pointer alignment to powers of two. This frees 18 bits in 64-bit pointers for encoding bounds information. BIMA demonstrates that on a clean-slate ISA design, fat pointers can be realized without a performance penalty, and a 3% memory overhead due to segmentation caused by alignment restrictions on BIMA pointers. *Low-fat-pointers* [68, 69] are an

alternative to fat pointers compatible with commodity 64-bit hardware architectures, such as x86-64. Low-fat-pointers require customized stack and heap allocators that restrict both stack frame and heap memory allocation sizes to a fixed finite set and split the main program stack and heap into several sub-stacks and sub-heaps, one for each possible allocation size. Pointer accesses are then validated according to the allocation bounds associated with the corresponding sub-stack or sub-heap. The improved compatibility comes at the cost of accuracy, as low-fat-pointers accesses are only enforced at allocation bounds. On average, low-fat-pointers add a performance penalty of 54% (16% for out-of-bounds writes) and memory overhead of 15% for stack data and incurs a 56% performance (13% for out-of-bounds writes) and 11% memory overhead for heap data.

Unfortunately, software-based memory safety protection incurs a significant amount of overhead, ranging from 48% to 116% [68, 69, 134, 135, 136]. Data-Flow Integrity also protects memory safety issues but incurs overhead ranging from 44% to 103% [29, 187]. On the other hand, PointGuard [49] has relatively low overhead (up to 20%) for protecting pointers, but PointGuard's memory-related assumption that attackers cannot read arbitrary memory is no longer practical. Thus, software-based pointer protection, in general, is not practical due to a high runtime overhead.

### 5.2.3 Hardware-based Defenses

Hardware-based solutions can reduce the overhead significantly. *HardBound* [66] can lower the overhead to 9% on average for pointers in C programs with architectural support. Intel's Memory Protection Extensions (MPX) incurs an average performance overhead of 50% due to the complexity of storing and loading bounds metadata. Fortunately, the ARM *pointer authentication* (PA) [158] offers a low cost (19.5% overhead for data pointer authentication [121]) and near-practical pointer authentication in the ARMv8-A processor architecture.

PA [158] is a hardware pointer authentication primitive introduced in the ARMv8.3-A processor architecture to protect programs from exploiting memory vulnerabilities. PA introduces a set of

new instructions for calculating and verifying a *Pointer Authentication Code* (PAC) for pointers. The use of an unauthenticated pointer would cause a memory translation fault. Each PAC is generated using a key from a set of five different keys and a tweakable modifier. The kernel generates the five keys for each process and stores them in internal CPU registers which are not accessible from userspace code. These keys remain the same throughout the process's lifetime. Out of the five keys, two (key a and key b) are used for generating PACs for code pointers, two (key a and key b) for data pointers and one (key a) for general purpose uses. The modifier usually captures the contexts of pointer declarations and accesses. For example, the instruction `pacia` creates the PAC for the address of an instruction (*i.e.*, code pointer) using key `a`. Similarly, the instruction `pacib` creates the PAC for an instruction address using key `b`. On the other hand, the instructions `pacda` and `autda` create and authenticate the PAC using the key `a` for a data pointer. To store PACs, PA uses the unused bits in the virtual address of 64-bit address space. In a 64-bit Linux kernel, PA uses 24 bits for the PACs, but the size can vary based on memory scheme and address tag usages.

### 5.2.4 Threat Model

Attackers' capabilities in this work are consistent with prior data-oriented attacks [35, 100, 101, 105, 172]. The key capability for a data-oriented attack is the capability to arbitrary memory read/write in an application's address space except for the code sections as this is prevented by Data Execution Prevention (DEP) security feature. In general, the attack model for data-oriented attacks is powerful because these attacks work in presence of all modern security features such as DEP [65], full or partial Relocation Read-Only, ASLR [38, 47, 52, 84, 97, 98, 108, 111, 199, 213], CPI [12, 49, 103, 104, 115, 126, 130, 158], CFI [24, 83, 124, 130, 132, 152, 222, 224], and memory protection [11, 53, 197, 215]. Since our prioritization framework works with underlying defenses (ARM PA [158], MPX [104], Softbound [134], AddressSanitizer [175], etc.), we assume these defenses are protected. We also attackers have no access to higher privilege levels. For example, the kernel stores the PA keys, so we assume that attackers cannot access the keys.

## 5.3 Data Object Prioritization

Figure 5.1 shows a high-level overview of our data object/pointer prioritization technique. The technique has two major parts: *i)* data object identification and tracking and *ii)* sensitive data object detection and prioritization. We utilize system or library I/O functions to identify and mark data objects (as tainted) that receive input from external sources (e.g., network, file system, and keyboard). We propagate the marked (i.e., tainted) data objects and propagate their values throughout a program. We use rule-based heuristics to identify sensitive tainted data objects. Finally, we prioritize sensitive data objects aiming to fine-tune security and performance of existing security mechanisms (e.g., asan [176], PA [158], MPX [104], Softbound [134], CETS [135], etc.).

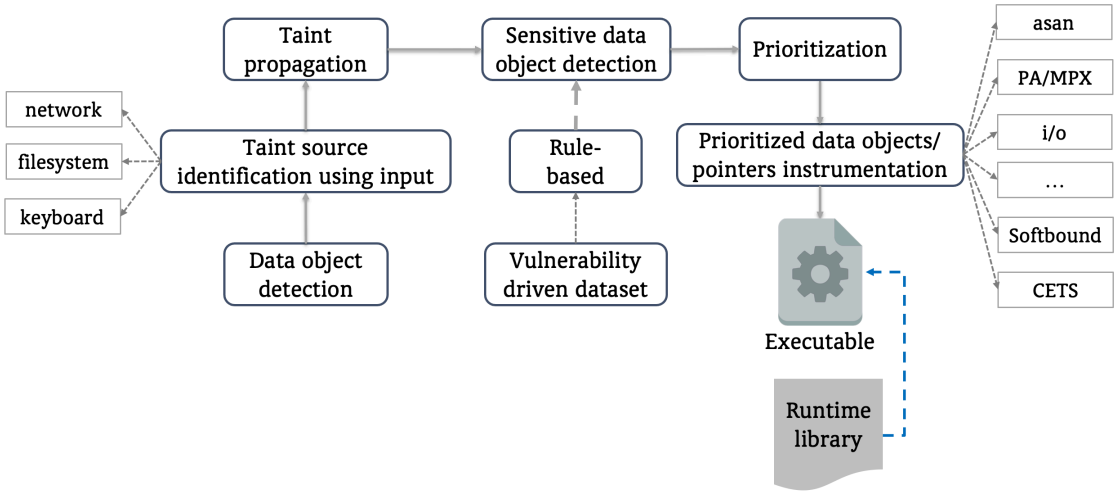



Figure 5.1: High-level overview of the data object prioritization technique.

### 5.3.1 Rule-based Prioritization

The data-oriented attacks in system security domains are evolving fast. Besides, attackers apply their distinct strategies to mount attacks. Oftentimes, the end-to-end strategies differ from each other. The unique attackers' capabilities make the anticipation of future attacks difficult. One way to capture existing attacks and anticipate future attacks is to break down advanced exploits and figure out the common components. These common components can help us design generic rules

and improve coverage for future attacks.

Our prioritization framework uses rule-based heuristics to detect sensitive data objects/pointers that could potentially lead to vulnerabilities. The rules are vulnerability and exploit driven, *i.e.*, we analyzed existing exploits and vulnerabilities to extract these rules. We identify the vulnerable data objects (and their pointers) from exploit program [35, 100], vulnerability description (CVE-2001-0820, CVE-2006-5815, CVE-2006-5815, CVE-2017-9430, CVE-2018-6151, CVE-2018-10111, CVE-2021-23017, etc.) exploit description [2], and literature [73, 173]. We identified two kinds of manipulation of data flow from these exploits/vulnerabilities: i) manipulation of data pointers, and ii) manipulation of data objects. We categorize the extracted seven rules into four categories.

Table 5.1: Simple rules to detect sensitive data objects and pointers.

| Rule # | Category | Short Description | Protection | Example CVE |
|---|---|---|---|---|
| Rule 1 | Control alteration | Data pointers used in predicates may alter program behavior | AT/MT | CVE-2006-5815 |
| Rule 2 | Control alteration | Data pointers used in loops may alter program flow or leak sensitive information | AT/MT | CVE-2006-5815 |
| Rule 3 | Proximity-based | Data pointers that are near to data buffers | AT/MT | CVE-2002-1496 |
| Rule 4 | Proximity-based | Data pointers used in vulnerable functions | MT | CVE-2021-31226 |
| Rule 5 | Erroneous | Data pointers that have been cast to different types | MT | CVE-2018-6151 |
| Rule 6 | Erroneous | Data pointers that have out-of-bound access | MT | CVE-2021-21773 |
| Rule 7 | Unguarded | Pointers that unbounded allocations | MT | CVE-2020-11612 |

**Rule 1** prioritizes data objects/pointers with predicate-use or p-use, i.e., whether a data object/-pointer has been used in any predicates. Listing 5.2 shows a real-world example where the manipulation of pointer *cp* and *pbuf* in the conditions at line 10 and 17 may change the program's flow.

**Rule 2** prioritizes data objects/pointers used in a loop condition and a loop body. Listing 5.3 shows an example from ProFTPD v1.3.0 where the manipulation of pointer *src* can control the execution of a loop. One can find DOP [101] gadgets by controlling the execution of the loop. A stack-based

[2]https://github.com/CyberGrandChallenge/samples

```
1 char *sreplace(char *s, ...) {
2       ...
3       char *src = s, *cp, **rptr;
4       char buf[BUF_MAX] = {'\0'}, *pbuf = NULL;
5       size_t rlen = 0, blen; cp = buf;
6       ...
7       while( *src ) {
8             ...
9             // replace a specifier with dynamic content stored in *rptr
10            sstrncpy(cp, *rptr, blen - strlen(pbuf));
11            if(((cp + rlen) - pbuf + 1) > blen){
12                  cp = pbuf + blen - 1; ...
13            } /* Overflow Check */
14            ...
15      }
16      ...
17      if((cp - pbuf + 1) > blen) { // off-by-one error
18            cp = pbuf + blen - 1; ...
19      } /* Overflow Check */
20      *cp++ = *src++;
21      ...
22 }
```

Listing 5.2: Usage of data pointer *cp* and *pbuf* in conditions at lines 10 and 17 in ProFTPD v1.3.0 (CVE-2006-5815)

buffer overflow (CVE-2006-5815) allows the overwrite of the data pointer *src* at line 3.

```
1 char *sstrncpy(char *dest, const char *src, size_t n) {
2       register char *d = dest;
3       for (; *src && n > 1; n--)
4             *d++ = *src++;
5 }
```

Listing 5.3: Loop manipulation for DOP [101] gadgets in ProFTPD v1.3.0 through data pointer overwrite (CVE-2006-5815)

It is important to mention here that data pointer manipulation can leak information about a program's address space by exploiting conditional branches or exploiting loops [73, 173].

**Rule 3** detects memory allocations that include an addressable buffer—typically an array—that is followed by a pointer. The exploitation of this pattern is attractive for two reasons. First, because the overflow and target are in the same buffer, there is no dependence on the overall memory layout of the program. And second, due to limitations of allocation-based bounds checking [134], such overflows can be performed even in the presence of common memory-protection schemes (such as the allocation-based AddressSanitizer [176]). Listing 5.4 shows an example of such an exploitable code pattern.

```
1 typedef struct mystruct_s {
2       char buffer[64];
3       __attribute__((signed)) int (*printer_func)(const char*, ...);
4 } mystruct_t;
5 void func(mystruct_t *ms) {
6       scanf("%s", ms->buffer);
7       ms->printer_func(ms->buffer);
8 }
```

Listing 5.4: A simplistic example of Rule 3

**Rule 4** prioritizes data objects/pointers used in vulnerable library functions such as strcpy(), memcpy(), gets(), strncpy(), sprintf(), etc. Listing 5.5 shows a vulnerability (CVE-2017-9430) in dnstracer v1.9 caused by strcpy function at line 6.

```
1 int main(int argc, char **argv) {
2       while ((ch = getopt(argc, argv, "4cCoq:r:S:s:t:v")) != -1) {...}
3       ...
4       if (argv[0] == NULL) usage();
5       // check for a trailing dot
6       strcpy(argv0, argv[0]);
7       if (argv0[strlen(argv[0]) - 1] == '.') argv0[strlen(argv[0]) - 1] = 0;
8       ...
9 }
```

Listing 5.5: Stack-based buffer overflow (CVE-2017-9430) in dnstracer v1.9 through strcpy function.

**Rule 5** prioritizes data pointers that have been cast from one type to another type. Listing 5.6 shows a bad cast in Google Chrome version prior to 66.0.3359.117 at line 2. At line 2, a *ChromeDownloadManagerDelegate* pointer is assigned after casting *download_manager->GetDelegate()* to *ChromeDownloadManagerDelegate**. Here, the *download_manager->GetDelegate()* returns a *DownloadManagerDelegate* pointer. But the *download_manager->GetDelegate()* can also return *DevToolsDownloadManagerDelegate* pointer through Chrome extensions. This bad cast vulnerability (CVE-2018-6151) allows attackers to perform an out-of-bounds memory read.

```
1 DownloadPrefs* DownloadPrefs::FromDownloadManager(DownloadManager* download_manager) {
2       ChromeDownloadManagerDelegate* delegate = static_cast<ChromeDownloadManagerDelegate*>(download_manager->
          GetDelegate());
3       return delegate->download_prefs();
4 }
```

Listing 5.6: Vulnerability due to bad casting in Google Chrome version prior to 66.0.3359.117

**Rule 6** prioritizes data objects/pointers where out of bound access may happen due to a set of

pointer arithmetic or due to incorrect indices. Listing 5.7 shows an example of out of bounds write in Nginx v0.6.18 - v1.20.0. Here, the out-of-bound write happens due to an off-by-one error (CVE-2021-23017). An extra dot ('.') character (i.e, 0x2E) can overwrite the least significant byte of the next heap chunk size metadata which may impact the size of the next heap chunk.

```
1 static ngx_int_t ngx_resolver_copy(ngx_resolver_t *r, ngx_str_t *name, u_char *buf, u_char *src, u_char *last) {
2     char            *err;
3     u_char          *p, *dst;
4     ssize_t          len;
5     ngx_uint_t       n;
6     ...
7     /* len is calculated using buf buffer with proper null byte check */
8     ...
9     len += 1 + n;
10    ...
11    dst = ngx_resolver_alloc(r, len);
12    name->data = dst;
13    n = *src++;
14    for ( ;; ) {
15        if (n & 0xc0) { // when processing the label
16            ...
17            n = *src++;
18        } else { // when processing a dot
19            ngx_strlow(dst, src, n);
20            dst += n; src += n;
21            n = *src++;
22            if (n != 0) { // this '.' will be extra when src points to "NUL Byte"
23                *dst++ = '.';
24            }
25        }
26    }
27 }
```

Listing 5.7: Out-of-bound write in Nginx v0.6.18 - v1.20.0

**Rule 7** identifies and prioritizes data objects/pointers that miss bound checking while being allocated, i.e., any data pointer that is allocated without any bound checking. Listing 5.8 shows an example of unbounded allocation at line 13 in the GEGL version through 0.3.32. This unbounded memory allocation leads to a denial of service (CVE-2018-10111).

### 5.3.2 Completeness and Representativeness of the Rules

As we discussed above, the unique attackers' capabilities make the anticipation of future attacks difficult. And, due to this difficulty of anticipating future attacks, it is challenging to guarantee the coverage and completeness of any rule-based heuristics that aim to capture present data-oriented

```
static gboolean render_rectangle (GeglProcessor *processor) {
    GeglCache   *cache = NULL;
    const Babl *format = NULL;
    gint pxsize;
    ...
    cache = gegl_node_get_cache (processor->input);
    format = gegl_buffer_get_format ((GeglBuffer *)cache);
    pxsize = babl_format_get_bytes_per_pixel (format);
    ...
    GeglRectangle *dr = processor->dirty_rectangles->data;
    ...
    guchar *buf; /* create a buffer and initialise it */
    buf = g_malloc (dr->width * dr->height * pxsize);
    g_assert (buf);
    ...
}
```

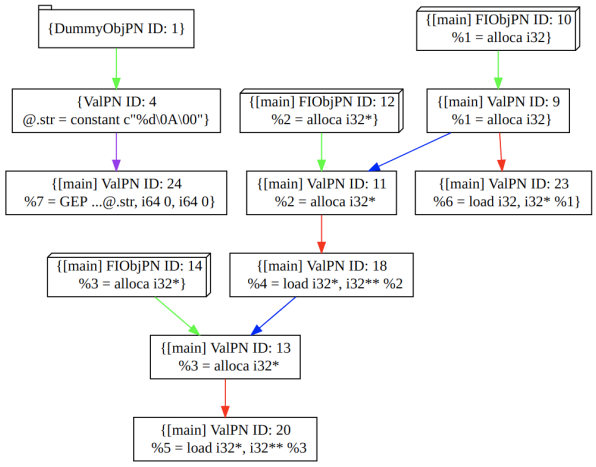
Listing 5.8: Unbounded allocation in GEGL version through 0.3.32 leads to denial of service (CVE-2018-10111)

```
#include <stdio.h>

void main() {
    int size = 10, *p, *q;
    p = &size;
    q = p;
    *q = 20;

    printf("%d\n", size);
}
```

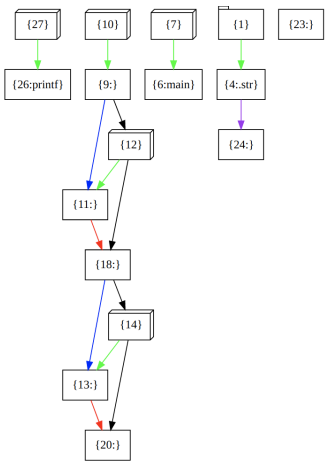
(a) Sample C program

```
%1 = alloca i32, align 4
%2 = alloca i32*, align 8
%3 = alloca i32*, align 8
store i32 10, i32* %1, align 4
store i32* %1, i32** %2, align 8
%4 = load i32*, i32** %2, align 8
store i32* %4, i32** %3, align 8
%5 = load i32*, i32** %3, align 8
store i32 20, i32* %5, align 4
%6 = load i32, i32* %1, align 4
%7 = i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0)
%8 = call i32 (i8*, ...) @printf(i8* %7, i32 %6)
ret void
```

(b) LLVM IR




(c) Program Assignment Graph (PAG)




(d) Constraint Graph

Figure 5.2: A sample program with LLVM IR, program assignment graph, and constraint graph.

attacks as well as anticipate future attacks. Similarly, a signature-based rule or end-to-end rule is not well-representative as a minor change in the attack pattern may not be covered by the existing rule. Thus, the representativeness of a rule depends on how readily the rule is applicable for various attacks. To achieve the coverage and representativeness of rules, we extract the rules by breaking down exploits and identifying key exploit strategies such as manipulation of conditions and loops through data pointers, utilizing the position of a data pointer (e.g., adjacent to a data buffer), utilizing vulnerable library functions, finding unbounded allocations and so on. Exploits may use a strategy or a combination of strategies from our identified strategies to construct their attacks. Our identified strategies are our best effort rules that tend to work well, as we experimentally confirmed in our evaluation 5.4. However, it will not be surprising to add new rules in our rule set to incorporate another class of future attacks.

We use the seven rules to flag/detect sensitive data objects/pointers from the tainted data objects/pointers. When a rule flags a pointer, our technique also flags the data object where the pointer points to. Two key requirements for tracking tainted values and applying our rules are i) pointer analysis, and ii) data/value flow graph. In the following section, we discuss the technique for constructing the data flow graph using pointer analysis.

### 5.3.3 Data Flow Construction

We apply the prioritization heuristics on top of a tainted data/value flow graph of a program. To construct the data flow graph, we utilize the interprocedural Static Value Flow (SVF) analysis [193]. SVF implements various pointer analysis techniques. We use Andersen's points-to analysis [7] for constructing our SVF graph. However, practical considerations must be made when choosing a points-to analysis technique considering the trade-offs between speed and precision. We discuss these practical considerations in Chapter 6. Next, we briefly discuss the construction of the SVF graph below. A short and long description of how SVF performs static value flow analysis on LLVM IR can be found in [195] and [193], respectively.

Since SVF performs its analysis on top of the LLVM IR, we first briefly discuss the key LLVM IR instructions. Figure 5.2 (b) shows the LLVM IR instructions of a sample C program in Figure 5.2 (a).

- **AllocaInst**. The alloca instruction allocates memory for local variables in a function.
- **StoreInst**. The store instruction writes the content of the first operand to the memory pointed by the second operand.
- **LoadInst**. The load instruction reads from memory pointed by the first operand.
- **GetElementPtrInst**. The getelementptr instruction reads a field or subelement of an aggregated data structure such as struct or array.
- **CallInst**. The call instruction calls a function.

SVF first converts LLVM IR instructions into a Program Assignment Graph (PAG). A node in a PAG is of two types: i) ValPN, and ii) ObjPN. ValPN represents an LLVM value (i.e., an IR pointer) and ObjPN represents an abstract memory object (i.e., the address-taken variable of an IR pointer). An edge in the PAG represents the constraints between nodes by capturing the address-of, load, store, and copy associations. For example, node 9 (rectangular shape) in the program assignment graph of Figure 5.2 (c) is a value pointer pointing to the object node 10 (hexagonal shape). The edge between node 9 and node 10 is an address-of constraint.

To perform pointer analysis, SVF starts with a copy of PAG. This copy of the PAG is called a constraint graph. SVF solves the constraints in the constraint graph by converting Load (red color edges) and Store (blue color edges) constraints to Copy constraints (black color edges) on each encounter of Load and Store instructions. The constraint graph in Figure 5.2 (d) is the final constraint graph after completing the constraint resolution. We can determine the points-to set from the final version of the constraint graph. For example, the points-set of node 11 in Figure 5.2 (d) is {12} and node 20 is {10} by following Load, Store, and Copy constraints from a node.

To construct Static Value Flow Graph (SVFG), SVF first annotates the potential use of a variable at loads, potential definitions and uses of the variable at stores, inter-procedural uses and definitions at call sites, and parameter passing/return at a function entries/exits, where the variable is pointed by a top-level pointer. SVF utilizes the points-to sets constructed in the previous steps using the constraint graph to find the points-to set of a top-level pointer. Finally, SVF constructs SVFG by converting all the address-taken variables to Static Single Assignment (SSA) form, merging multiple definitions using `phi` instructions, and connecting the definition-uses for each SSA variable. Figure 5.4 shows the SVFG of an example program in Figure 5.3. We utilize this example program to demonstrate our taint analysis and rule design.

The example program (Figure 5.3) has five memory objects (two local, two dynamic, and one global). The hexagonal shapes (nodes 1, 5, 13, 32, and 39) in the SVF graph (Figure 5.4) show the memory object nodes. The rectangular nodes show how the values of the memory objects flow through different IR instructions.

### 5.3.4 Taint Analysis

We design our rules on top of the tainted SVF graph. We identify the taint sources and propagate the tainted data through the SVF graph. The seven rules detect vulnerable tainted data (i.e., taint sinks).

**Identification of tainted sources**. Vulnerable data objects or pointers must be externally manipulatable. To manipulate any data in a program, attackers must use the input channels/streams (e.g., network, file system, or keyboard) to pass their crafted commands or packets. Thus, the input channels are the intuitive sources for taint analysis. In most cases, the input sources are standard Glibc library functions such as *read*, *recv*, *getc*, *recvmsg*, *scanf*, *fscanf*, *fread*, *fgets*, etc. However, sometimes applications use some wrapper functions of the standard library functions. Identification of these wrapper functions can make analysis faster. We analyze nine real-world programs and twenty DARPA Cyber Grand Challenges to identify the library functions as well as the wrapper

functions around those library functions. We call these functions input reading functions. We also consider the *main* function as an input reading function. We start the tainting process from each parameter and return-value (i.e., if a value is returned) of an input reading function if the parameter or returned value is a memory object/pointer. If a parameter or returned value is a pointer, then the tainting process starts from the points-to set of the pointer parameter or returned value.

```
1  #define MAX_SIZE 11
2  int global_array[100] = {-1};
3
4  void testcase(int ts) {
5      char buf[10];
6      //if (ts > MAX_SIZE) exit(0);
7      char *s = (char *) malloc(ts * sizeof(char));
8      for(int i=0; i < ts; i++) s[i] = i + '0';
9      if (s[0]=='a') printf("%d\n", s[ts-1]);
10     fscanf(stdin, "%s", s); //'s' marked as tainted
11     memcpy(buf, s, strlen(s)); // sink
12     printf("%s\n", buf);
13 }
```

```
10 int heap (int argc) {
11     int *array = (int *)malloc(sizeof(int) *100);
12     int res = array[argc + 100];  // overflow
13     free(array);
14     return res;
15 }
16 int main(int argc, char **argv) {
17     int size;
18     scanf("%d", &size); //'size' marked as tainted
19     testcase(size);
20     int r = heap(argc);
21     printf("%d", r);
22     return global_array[argc + 100]; //overflow
23 }
```

Figure 5.3: Motivating example (C program)

**Propagation of tainted data**. We propagate the tainted data through the SVF graph simply by traversing all successors from a tainted node. However, we need to address the following limitations of the SVFG for the completeness of the taint propagation process.

- First, SVF does not capture the dependency between a dynamically allocated object and an argument of an allocation function.
- Second, it does not add a relationship between an array/object and an index/pointer when the array or object is accessed through the index or pointer arithmetic.
- Third, it does not track when a function (e.g., *memcpy*) stores value from one parameter to another.

We notice some of these limitations in the SVF graph in Figure 5.4. In the example C program (Figure 5.3), the *malloc* call uses *ts* as its argument at line 7. Since *ts* has been propagated from

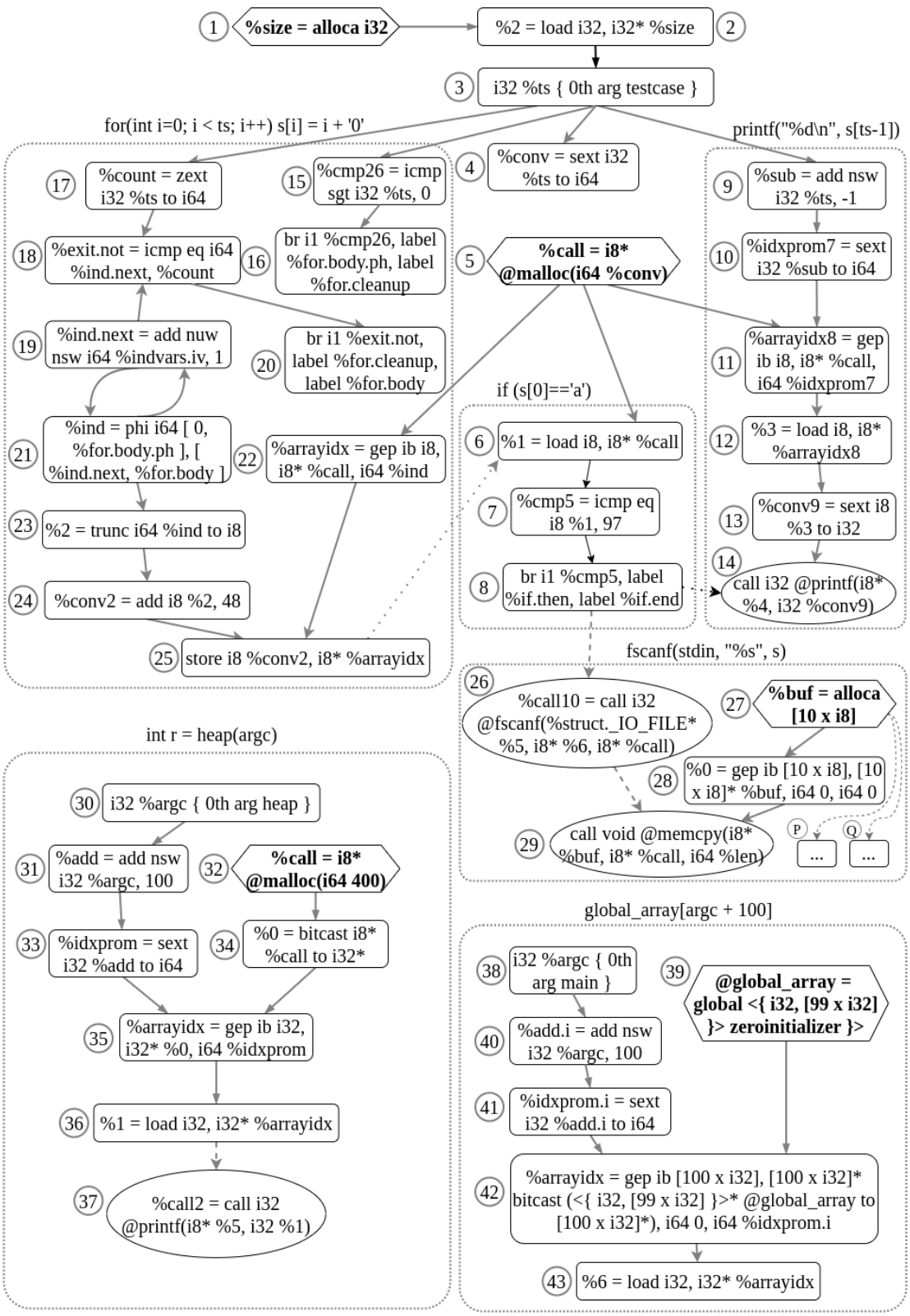



Figure 5.4: Static Value Flow Graph (SVFG) of the motivating example in Figure 5.3. *gep ib* → getelementptr inbounds.

*size*, it is tainted. Thus, the dynamic object created by *malloc* needs to be tainted. However, there exists no way to taint the *malloc* (node ❺) by propagating the tainted data from node ❶ in the SVF graph (Figure 5.4).

We notice the second limitation in the SVF graph for lines 18 and 19 in the example program (Figure 5.3). The *malloc* call at line 18 uses a constant value as its argument, but the malloced object is accessed using the *argc* variable. Since *argc* is tainted (due to as an argument of *main* function), the malloced object needs to be tainted. In this case, we also notice that we are unable to propagate the tainted value from node ㉚ (i.e., *argc*) to node ㉜ (i.e., the *malloc* call) in the SVF graph).

The node ㉙ in the SVF graph shows the third limitation where the *memcpy* function copies the *malloced* object *%call* to *%buf*. A simple traversal through the SVF graph nodes does not show the dependency between node ㉖ and node ㉗. Beside, the dependency between node 26 and any successor nodes (node Ⓟ and Ⓠ) of node ㉗ is also not captured.

To address the first two limitations, we identify all the address-taken memory object nodes and GEP instruction nodes in the SVFG with one or more tainted operands of these nodes and start/continue the tainting process from/through these nodes. We maintain a data structure to store the node ID for all tainted nodes. We utilize this data structure to identify address-taken memory object nodes and GEP instruction nodes with tainted operands. We discard the first operand of GEP instruction as we aim to identify whether the second or subsequent operands make the first operand tainted. The address-taken nodes in the SVFG (Figure 5.4) are ❶, ❺, ㉗, ㉜ and ㊴ where only node ❺has a tainted operand (i.e., *%conv*). So, we taint node ❺ and continue the tainting process through node ❺. All the GEP instruction nodes in the SVFG are ⓫, ㉒, ㉘, ㉟, and ㊷. We discard node ㉘ because the GEP instruction of this node has no tainted operand. For the rest nodes, we obtain the points-to set of the first operand of a GEP instruction and start the tainting process from each node of the points-to set if the node is not already tainted. For example, the points-to set of the first operand of GEP instruction in node ㉟ is {32}. So, we start the tainting process from node ㉜.

We address the third limitation by identifying all function call instructions that store the value of their second parameter to the first parameter. Such function call instructions include *memcpy*, *memmove*, *strcpy*, *strncpy*, *strcat*, and their variations. If the second parameter is tainted, then we taint the first parameter and its points-to set. One such function call in Figure 5.4 is *memcpy* (node 29). Since the second parameter of the *memcpy* is tainted, we update the taint list by starting the tainting process from the first parameter (i.e., *%buf*) and its points-to set (in this case *%buf*).

Algorithm 5.5 shows the pseudocode of the tainting process. Lines 3–8 in the algorithm identify the taint sources and start the taint propagation from those sources. Lines 9–12 propagate tainted data through dynamically allocated objects if the argument of allocation function is tainted. Lines 13–17 taint all base objects (i.e., points-to set of operand zero) of a GEP instruction if any operands (starting from the second operand) of GEP are tainted. Finally, lines 18–24 check if any function copies values from its second argument to the first argument, if so, then taint the first argument as well as its points-to set. The *UpdateTaintList* function (lines 18—25) traverses all the reachable nodes starting from a node in the SVFG.

**Identification of sinks**. We identify seven sink-types (i.e., sensitive uses of data object/pointer) using our rules in Table 5.1. We discuss the sink identification in the next section (Section 5.3.5).

### 5.3.5 Rule Design

The complete process of identifying one sink type is different from another. However, the identification processes share some steps. To determine sinks using Rule 1, 2, 4, and 5, we first identify the tainted address-taken memory objects from the SVF graph considering only the object nodes whose object-types are pointer types. Then, we obtain the set of pointers for each identified object using the pointer analysis result. Note that this pointer set is different from a points-to set as a points-to set contains a set of objects whereas the pointer set in this case contains a set of pointers whose points-to sets contain the given object. For example, one of the tainted object-nodes from the SVFG in Figure 5.4 is node ❺. The pointer set pointing to node ❺ is {5, 11, 22}. We utilize

**Algorithm 1:** Identifying taint sources and propagating the tainted data

```
Function PerformTaintAnalysis(PAG, SVFG):
1   InputFuncs ← a set of all input reading functions
2   TaintedNodes ← {}
3   foreach Callee callee : PAG.getCallSites() do
4       if InputFuncs.contain(callee) then
5           foreach Argument arg: callee.getArguments() do
6               UpdateTaintList(arg, TaintedNodes)
7               foreach Object target: points-set(arg) do
8                   UpdateTaintList(target, TaintedNodes)
9   foreach AddressTakenNode node: SVFG.nodes() do
10      hasTaintedArg = node.hasAnyTaintedArg()
11      if hasTaintedArg is true then
12          UpdateTaintList(node, TaintedNodes)
13  foreach GEP node: SVFG.nodes() do
14      isTainted = node.checkForTaintedArg(startIndex=1)
15      if isTainted is true then
16          foreach Object target: points-set(node.getOperand(0) do
17              UpdateTaintList(target, TaintedNodes)
18  foreach CallInst node: SVFG.nodes() do
19      isCopyFunc = is2ndArgCopiedTo1stArg(node)
20      if isCopyFunc is true then
21          operand2 = node.getOperand(1)
22          UpdateTaintList(operand2, TaintedNodes)
23          foreach Object target: points-set(node.getOperand(0) do
24              UpdateTaintList(target, TaintedNodes)

Function UpdateTaintList(arg, TaintedNodes):
25  worklist ← {}
26  worklist.insert(arg)
27  while worklist not empty() do
28      node = worklist.pop()
29      TaintedNodes.insert(node)
30      foreach Successor successor: node.successors() do
31          if successor not visited then
32              workilist.insert(successor)
```

Figure 5.5: Taint source identification and propagation

the LLVM built-in definition-use chains for the nodes in this pointer set to construct the usage list of node ❺ which is {5, 6, 7, 8, 11, 12, 13, 14, 22, 25, 26}. Once we construct this usage list, we

apply the following techniques to capture sinks using Rule 1, 2, 4, and 5.

- To capture sinks using Rule 1, i.e., usage of data pointers in predicates, we check if any node from the usage list has a compare instruction.
- To capture sinks using Rule 2, i.e., usage of data pointers in loops, we check if any node from the usage list has a load/store/compare instruction and has been used in a loop's predecessor, header, and latch. We get the loop information from an IR module using the LLVM *LoopAnalysis* [3] pass.
- We capture the sinks using Rule 4, i.e., usage of data pointers in vulnerable functions, by checking if any arguments of a vulnerable function are from the usage list.
- We check if any node from the usage list has a *bitcast* instruction to check sinks using Rule 5 to find out incompatible casting of data pointers. LLVM IR uses the *bitcast* instructions to convert one data type to another. We filtered out all the trivial casting like *char* * to anything or vice-versa and non-pointer casts. We check the incompatibility of bitcast's operands by determining the allocation size of each operand's pointed type using the data layout of an LLVM module.

We detect the sinks related to data pointers that are closed to a data buffer (Rule 3) by traversing all the tainted *Alloca* instructions in a function and all the tainted global variables to see if any pointer follows a tainted data buffer. If so, we mark the pointer as sensitive.

To detect sinks related to out-of-bound access (Rule 6), we first obtain all the tainted objects and pointers. We then apply three optimizations to filter out safe tainted objects/pointers. First, we only need the object/pointer operands from Load, Store, and Call SVF nodes as these are the nodes that deal with memory accesses. Second, we apply the stack safety analysis [4] to filter out safe allocated variables that are free from memory access bugs. And finally, we statically approximate

[3] https://llvm.org/doxygen/classllvm_1_1LoopAnalysis.html
[4] https://llvm.org/docs/StackSafetyAnalysis.html

the size and offset of the rest operands using LLVM's ObjectSizeOffsetVisitor [5] class. We flag the rest operands as sensitive.

To obtain the sinks using Rule 7, i.e., unbounded memory allocations, we first determine the dynamic memory allocation nodes from the SVF graph using SVF's memory allocation APIs. We filter out the allocation nodes that are untainted and are in dead functions. Once we get the allocation sites or nodes in the SVF graph, we obtain the corresponding nodes in the Interprocedural Control-Flow Graph (ICFG). Figure 5.6 shows the partial ICFG for the *testcase* function where node ⓫ is the ICFG node for node ❺ in the SVFG (Figure 5.4). We perform a backward search from the obtained ICFG nodes to fetch *cmp* instructions. We follow multiple search paths to extract *cmp* instructions. The goal is to check if the argument used in a memory allocation function is bounded. It is straightforward to search backwardly for *cmp* instructions. However, there are three key challenges here: i) how to deal with path explosion, ii) how to deal with loops in the ICFG, and iii) how to determine that a *cmp* instruction is the relevant *cmp* instruction that is related to the argument of a memory allocation function.

To address the first challenge, we use two parameters to avoid the problem of path explosion. The first and second parameters dictate how many paths to explore and how far to explore in a path, respectively. We can tune these parameters to balance between performance and precision of our analysis. To address the second challenge, we modify the ICFG by removing all the edges that create loops in the graph. We address the third limitation by determining if a *cmp* instruction and the argument of an allocation function have a common ancestor in the SVF graph.

To find the common ancestor, we first obtain all the SVF nodes that are backwardly *reachable* from an allocation's argument node. The argument of SVF node ❺ in Figure 5.4 is *%conv* (node ❹). For convenience, we extract the partial SVF graph in Figure 5.7. The *reachable* nodes from node ❹ in the partial SVF graph are node ❷, node ❸, node ❹', and node ❹. We then take each *cmp* node from the search paths that we have already obtained from the ICFG, obtain its corresponding node(s)

[5] https://llvm.org/doxygen/classllvm_1_1ObjectSizeOffsetVisitor.html

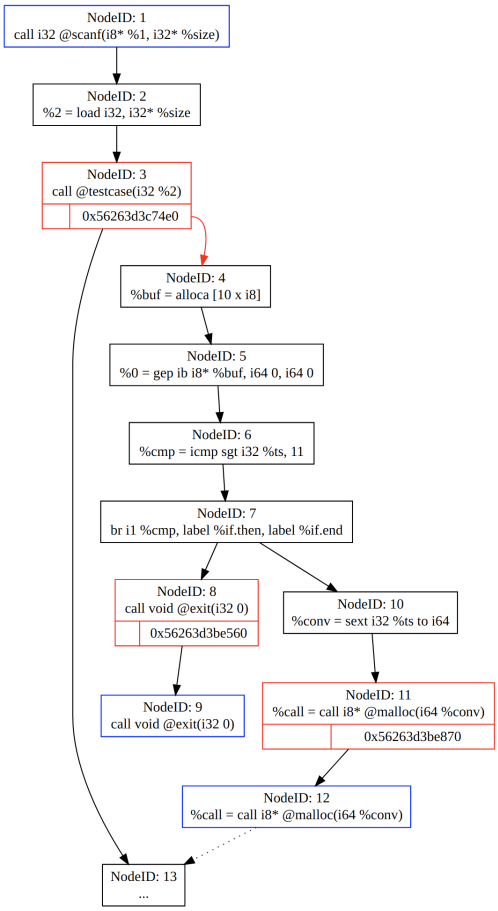



Figure 5.6: Interprocedural Control-Flow Graph (ICFG) for *testcase* function in Figure 5.3.

from the SVF graph, and traverse backwardly from the corresponding node(s). If the backward traversal encounters any node from the *reachable* nodes, that means the *cmp* node has a common ancestor with the argument node of an allocation function, hence this *cmp* node is related to the argument of the allocation function. In Figure 5.7, the common ancestor of node ID 4 and node ID 15 is node ID 3.

It is important to mention that the symbolic execution [110] can improve the precision of Rule 7. Our approximation is to confirm the presence of a bound condition. But sometimes the bound conditions can be wrong or have one type of bound checking (i.e., lower or upper bound checking). This wrong bound condition or one-way bound checking is the common source of many real-world vulnerabilities (e.g., CVE-2006-5815 and CVE-2021-23017). Our rule along with the symbolic execution can precisely determine if any allocation is correctly bounded.

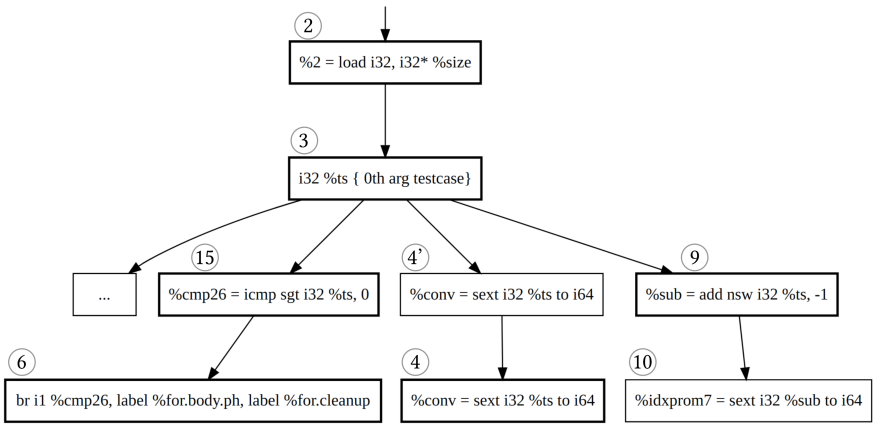



Figure 5.7: Partial SVF graph. Common ancestor of node 4 and node 15 is node 3.

## 5.4 Evaluation

### 5.4.1 Implementation

We implemented our analysis in LLVM 12 [6]. To perform the pointer analysis and data-flow construction, we utilize the latest version of SVF [193]. SVF also uses LLVM 12. We performed our analysis on top of the LLVM bitcode. We obtained a single whole program bitcode file using the whole program LLVM tool [7]. We have implemented ten LLVM analysis passes to perform taint tracking, i.e, taint source identification, taint propagation, and taint sink detection. We have also implemented one instrumentation pass to instrument LLVM IR to include ARM pointer authentication instructions. Besides, we have modified the Address Sanitizer tool [175] to add an option to instrument data objects/pointers prioritized by our rules. We will open our source code to the public upon acceptance of this work.

### 5.4.2 Prioritization Effectiveness

We evaluate the effectiveness of our prioritization technique by flagging/prioritizing vulnerable data objects. A requirement for this evaluation is ground truth vulnerable data objects. To the best

[6] https://releases.llvm.org/12.0.0/docs/ReleaseNotes.html
[7] https://github.com/travitch/whole-program-llvm

of our knowledge, there exists no such dataset. There exists datasets such as DARPA Cyber Grand challenges [1] and Software Assurance Reference (SAR) [144] that have annotations of the locations of vulnerabilities. We construct the ground truths of vulnerable data objects by identifying 33 vulnerable data objects from 18 programs including 5 real-world applications and 10 test cases from SAR dataset. We extracted six (6) vulnerable data objects from five (5) vulnerable server programs where attackers have demonstrated data-oriented exploits. We identified 18 vulnerable data objects from 13 DARPA Cyber Grand challenges and 10 data objects from SAR dataset, where the data objects are related to a vulnerability in data buffers or pointers. The SAR dataset [144] contains thousands of test cases by adding minor modifications to a base test case in a category of Common Weakness Enumeration (CWE). We select 10 vulnerable objects from these base test cases combining C and C++ sources as well as covering five CWE categories. We identify the vulnerable data objects from exploits for *ghttpd-1.4*, *wu-ftpd-2.6.0*, *proftpd-1.3.0*, *httpd-2.4.49*, and *nullhttpd-0.5.0* through exploit program [35, 100], exploit description [8], and literature [73, 173]. The DARPA Cyber Grand challenges and SAR dataset have labels for the location of the vulnerabilities. We manually analyze each challenge to map the vulnerable data objects/pointers to their defined, allocation and used locations in the respective programs. Table 5.2 shows the vulnerable data objects/pointers extracted from different applications/programs. We have excluded the allocation source code for some data objects from the table for brevity.

Table 5.3 shows the 18 programs and 10 test cases in our evaluation having a total of 13,120 data objects. Out of these 13,120 data objects, the five real-world programs contain 12,591 data objects while the DARPA challenges contain 208 and the SAR dataset contains 321. According to the result in Table 5.3, our prioritization technique prioritizes 2,324 ($\sim$18%) data objects on average, out of 12,591 data objects from real-world applications. That means the rules filter out 82% of data objects on average that do not need protection. The filtration rate is much lesser (25%) for the DARPA CGC challenges compared to the server programs due to the size of the challenge programs. The CGC challenges are small programs with vulnerabilities designed for evaluating

[8] https://github.com/CyberGrandChallenge/samples

Table 5.2: Vulnerable data objects/pointers in various programs with their definition functions and line numbers. The * in some source file names and function names indicate that parts of the file name or function name have been truncated for space.

| **Ground truth data object** | **Application** | **Source** | **Defined function** | **Line #** |
|---|---|---|---|---|
| char *ptr | ghttpd-1.4 | protocol.c | serverconnection | 62 |
| struct passwd *pw | wu-ftpd-2.6.0 | ftpd.c | global | 264 |
| char *src | proftpd-1.3.0 | src/support.c | sreplace | 631 |
| char *cp | proftpd-1.3.0 | src/support.c | sreplace | 631 |
| apr_array_header_t *log_format; | httpd-2.4.7 | server/log.c | log_error_core | 1209 |
| char *pPostData; | nullhttpd-0.5.0 | src/http.c | ReadPOSTData | 92 |
| char first[32]; | CROMU_00003 | yolodex.c | struct _contact | 41 |
| char last[32]; | CROMU_00003 | yolodex.c | struct _contact | 42 |
| char phone[16]; | CROMU_00003 | yolodex.c | struct _contact | 43 |
| char name[MAX_NAME_LEN]; | CROMU_00039 | packet.c | HandleReadRequest | 155 |
| char buf[64]; | KPRCA_00011 | main.c | login | 286 |
| int *func_args | KPRCA_00013 | accel.c | infixtorpn | 482 |
| question_t *cur | KPRCA_00023 | form.c | handle_update | 258 |
| cgcf_Shdr *shdr = NULL; | KPRCA_00037 | main.c | main | 170 |
| writer_t *writer | KPRCA_00040 | main.c | decompress | 637 |
| char output[MAX_DATA_SIZE] | KPRCA_00047 | main.c | perform_ocr | 304 |
| locker_t *locker | KPRCA_00050 | vault.c | store_in_vault | 140 |
| char *dp | KPRCA_00056 | service.c | execute_program | 157 |
| unsigned char **rot_table | KPRCA_00064 | sc.c | sc_bwt | 248 |
| unsigned char *out | KPRCA_00064 | sc.c | sc_bwt | 249 |
| char* message_buf=NULL | NRFIN_00033 | service.c | auth_failure | 74 |
| char* message_buf; | NRFIN_00033 | service.c | auth_success | 105 |
| char **lines | NRFIN_00042 | viewscript.c | run_viewscript | 603 |
| int buffer[10] | CWE121*/s01/ | *_socket_01.c | *_socket_01_bad | 107 |
| int buffer[10] | CWE121*/s01/ | *_fgets_01.c | *_fgets_01_bad | 44 |
| int buffer[10] | CWE121*/s01/ | *_fscanf_01.c | *_fscanf_01_bad | 31 |
| int * buffer | CWE122*/s01/ | *_socket_01.cpp | bad | 110 |
| int * buffer | CWE122*/s01/ | *_fgets_01.cpp | bad | 47 |
| int * buffer | CWE122*/s01/ | *_fscanf_01.cpp | bad | 34 |
| int buffer[10] | CWE126*/s01/ | *_fgets_01.c | *_fgets_01_bad | 43 |
| int * buffer | CWE134*/s01/ | *_fprintf_01.c | *_socket_fprintf_01_bad | 49 |
| char * myString; | CWE789*/s01/ | *_socket_01.c | *_socket_01_bad | 118 |
| char * myString; | CWE789*/s01/ | *_char_fgets_01.cpp | bad | 58 |

different techniques that aim to capture or fix the vulnerabilities. Similarly, the SAR test cases are small programs designed to evaluate accuracy of a static analysis by adding a minor medication on top of a base program. Also, the SAR test cases have a small number of data objects that have external dependencies. And, all the test cases have the same or similar format. That is why we observe the similar number of data objects and prioritized data objects for the SAR test cases.

We rank the prioritized data objects for each application and test cases. To rank the data objects, we use the number of rules that mark a data object as sensitive or vulnerable as a metric and use the metric to rank all the data objects in an application in a decreasing order making the top data objects the most sensitive.

The actual effectiveness of the prioritization technique relies on how effectively our technique prioritizes vulnerable data objects. To measure how well our technique prioritizes vulnerable data objects, we tested 33 ground truth vulnerable data objects (column one in Table 5.3) to see if our prioritization technique flags and ranks those 33 data objects. Out of the 33 ground truth data objects, our technique flagged 32 data objects as sensitive. We also determined how well our technique ranks these 32 vulnerable data objects.

The fifth column of table 5.3 shows the top k number of prioritized data objects that include the ground truth data object(s). For example, the top four prioritized data objects in *ghttpd-1.4* server include the ground truth data object pointed by *char *ptr*.

We computed the percentages of this top k number with respect to the prioritized data objects and all data objects in the sixth and seventh columns, respectively. On average, we noticed that all ground truth data objects can be found in the top 11% of the prioritized data objects for real-world server applications. These top 11% data objects with respect to the prioritized data objects are only 3% of all data objects for an application, on average. For DARPA CGC challenges, we found the ground truths are in the top 25% of the prioritized data objects where the 30% data objects are 24% with respect to all data objects. The top k percentage is higher (60%) for SAR dataset compared to real-world or DARPA challenges. The reason is that the SAR test cases have a few objects with external dependencies indicating a few number of objects to prioritize. However, our rules successfully flagged and ranked seven vulnerable data objects as the top one out of the ten data objects.

These results suggest that we may need to protect as low as only around 3% of all data objects on average for a real-world application. We can decrease or increase this percentage with a trade-

Table 5.3: The number and percentage of top k prioritized objects needed for detecting/flagging ground truth data objects in 18 programs including five real-world server applications and 10 test-cases from SAR dataset.

| **Ground Truth (GT) data object (obj)** | **Program name** | **All data objs** | **Prio. data objs** | **GT in top-k prio. objs** | **Percent of top-k in prio. objs** | **Percent of top-k in all objs** |
|---|---|---|---|---|---|---|
| **Real-world applications** | | | | | | |
| char *ptr | ghttpd-1.4 | 77 | 14 | 4 | 24% | 5% |
| struct passwd *pw | wu-ftpd-2.6.0 | 590 | 378 | 34 | 10% | 6% |
| char *src | proftpd-1.3.0 | 5070 | 1313 | 236 | 18% | 5% |
| char *cp | | | | 28 | 2% | 1% |
| apr_array_header_t *log_format; | httpd-2.4.7 | 6754 | 587 | 77 | 13% | 1% |
| char *pPostData; | nullhttpd-0.5.0 | 100 | 37 | 1 | 2% | 1% |
| | **Sum.→** | **12591** | **2324** | **Avg.→** | **11%** | **3%** |
| **DARPA CGC challenges** | | | | | | |
| char first[32] | CROMU_00003 | 15 | 12 | 2 | 17% | 13% |
| char last[32] | | | | 2 | 17% | 13% |
| char phone[16] | | | | 2 | 17% | 13% |
| char name[MAX_NAME_LEN]; | CROMU_00039 | 8 | 8 | 6 | 75% | 75% |
| char buf[64]; | KPRCA_00011 | 28 | 19 | 10 | 53% | 36% |
| int *func_args | KPRCA_00013 | 38 | 36 | – | – | – |
| question_t *cur | KPRCA_00023 | 12 | 12 | 1 | 8% | 8% |
| cgcf_Shdr *shdr = NULL; | KPRCA_00037 | 9 | 9 | 2 | 22% | 22% |
| writer_t *writer | KPRCA_00040 | 6 | 5 | 1 | 20% | 17% |
| char output[MAX_OCR_DATA_SIZE] | KPRCA_00047 | 9 | 8 | 6 | 86% | 67% |
| locker_t *locker | KPRCA_00050 | 10 | 7 | 1 | 10% | 10% |
| char *dp | KPRCA_00056 | 6 | 5 | 1 | 20% | 17% |
| unsigned char ** rot_table | KPRCA_00064 | 23 | 20 | 12 | 60% | 52% |
| unsigned char *out | | | | 3 | 15% | 13% |
| char* message_buf= NULL | NRFIN_00033 | 11 | 6 | 2 | 22% | 18% |
| char* message_buf | | | | 1 | 11% | 9% |
| char **lines | NRFIN_00042 | 33 | 4 | 1 | 25% | 3% |
| | **Sum.→** | **208** | **156** | **Avg.→** | **30%** | **24%** |
| **SAR dataset** | | | | | | |
| int buffer[10] | CWE121/s01/socket_01 | 31 | 2 | 2 | 100% | 6% |
| int buffer[10] | CWE121/s01/fgets_01 | 32 | 2 | 1 | 50% | 3% |
| int buffer[10] | CWE121/s01/fscanf_01 | 31 | 2 | 1 | 50% | 3% |
| int * buffer | CWE122/s01/socket_01 | 32 | 2 | 1 | 50% | 3% |
| int * buffer | CWE122/s01/fgets_01 | 33 | 2 | 1 | 50% | 3% |
| int * buffer | CWE122/s01/fscanf_01 | 32 | 2 | 1 | 50% | 3% |
| int buffer[10] | CWE126/s01/fgets_01 | 32 | 2 | 2 | 100% | 6% |
| char *data | CWE134/s01/fprintf_01 | 32 | 2 | 2 | 100% | 6% |
| char * myString; | CWE789/s01/socket_01 | 32 | 4 | 1 | 25% | 3% |
| char * myString; | CWE789/s01/fget_01 | 33 | 4 | 1 | 25% | 3% |
| | **Sum.→** | **321** | **24** | **Avg.→** | **60%** | **4%** |

off between performance and security. If we prioritize security, then we need to increase this percentage, and otherwise for performance.

**False positives and false negatives**. Our prioritization technique generates a false positive data object/pointer when a data object/pointer is not sensitive, but our prioritization technique flags and prioritizes the object/pointer. Similarly, when a data object/pointer is sensitive, but our prioritization technique does not flag and prioritizes the object/pointer, our technique generates a false negative.

To find out the number of false positives, we manually analyzed all the prioritized data objects from the DARPA CGC challenges and SAR dataset. However, it is very challenging to manually analyze more than 2k prioritized data objects from the five real-world programs (i.e., *wu-ftpd-2.6.0*, *proftpd-1.3.0*, *httpd-2.4.7*, and *nullhttpd-0.5.0*). To make our analysis easier, we pseudo-randomly selected 12 data objects from each of the five applications. That means we randomly selected some source files from an application and analyzed several data objects from that source file. This makes our manual analysis easier and accelerated. The second challenge is to precisely identify sensitive data objects by manual analysis due to complex value flow through pointers. To approximate the sensitiveness of a data object, we set two criteria: (i) external dependency and (ii) necessary conditions to fall under at least one rule. If a prioritized data object does not satisfy these two criteria, we marked it as a false positive. In our evaluation, our prioritization technique has zero false positives in our tested data objects based on our criteria.

However, one can argue that the technique discussed above will always have zero false positives as the technique will always find the two criteria. It just reassures the correctness of the taint analysis and rule enforcement. We understand this limitation and leave it as future work. One direction could be the use of symbolic execution to figure out the bounds of an object, and identify the safe access (i.e., read/write operation) of the object. If our technique prioritizes such objects, then these objects will be considered as false positives. It is also important to note here that over approximating is fine in this work, i.e., prioritizing some non-sensitive data objects/pointers does

not hurt as long as the prioritization technique does not miss any sensitive objects/pointers.

Our technique missed one data object from the ground truth data objects, i.e., one false negative out of 23 ground truths. This data object (*int *func_args* is from one of the DARPA CGC challenges (*KPRCA_00013*). The reason for this false-negative case is due to operational dependence. That means the data object is not tainted, but operationally dependent on a tainted value.

Listing 5.9 shows an example of such operational dependence. In the listing, the reallocation of *func_args* at line 9 depends on the condition at line 7. A specific case of the variable *arg_type* (i.e., case FUNCTION at line 6) increases the index variable *func_idx* at line 14. However, there is no data-flow dependency between *arg_type* and *func_idx*. This makes the *func_args* data object only operationally dependent with *arg_type*.

```
1  switch (arg_type) {
2      case DOUBLE:
3      case CELL_ID:
4          enqueue_copy(&output_q, arg, strlen(arg) + 1);
5              break;
6      case FUNCTION:
7          if(func_idx == func_size) {
8              func_size *= 2;
9              int *temp = realloc(func_args, func_size * sizeof(int));
10             if (temp == NULL)
11                 goto error;
12             func_args = temp;
13         }
14         func_args[++func_idx] = 0;
15         push_copy(&operators, arg, strlen(arg) + 1);
16         break;
17     case BAD_CELL:
18         break;
19     default:
20         goto error;
21 }
```

Listing 5.9: False negative case – operational dependence

One can raise the concern of the completeness of our ground truths, i.e., how to make sure that the

ground truths include all the sensitive or vulnerable data objects that an application can have. If we can confirm the inclusion of all vulnerable data objects in our ground truths, it can make our false-negative analysis imprecise. This is also a limitation of our current evaluation. To solve this limitation, we plan to use fuzzers for automating the ground-truth finding process. The idea is to apply fuzzers to generate crashes. If the crashes are related to a data object or its pointers, then we can mark the data object as sensitive. We also leave this as future work.

Our findings in Table 5.3 in terms of identifying and prioritizing the ground truths indicate empirical guarantee as opposed to a theoretical guarantee. We can claim a theoretical guarantee if we can prove the completeness of our rules. However, proving the completeness of the rules is a challenging task as the attacks in the system security domain are constantly evolving. We provide the empirical guarantee through our best effort approach.

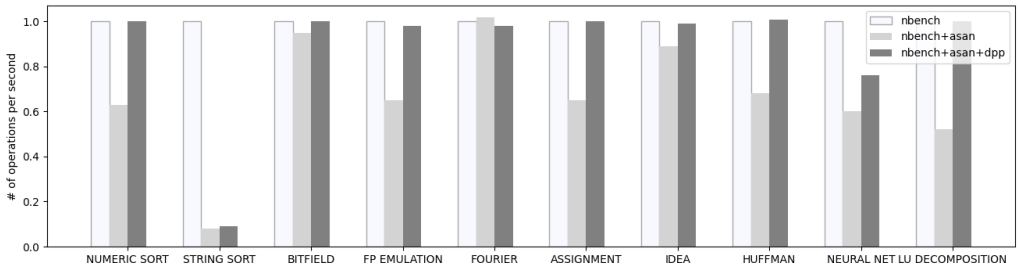



Figure 5.8: The number of operations per second in three scenarios after normalizing with the baseline (i.e., nbench)

### 5.4.3 Performance Evaluation

We utilize two approaches for evaluating the impact of our prioritization mechanism on performance: i) through AddressSanitizer [9] tool and ii) ARM Pointer Authentication mechanism [158]. AddressSanitizer detects memory-related errors by instrumenting data objects by inserting extra code and enforcing the instrumented code through a runtime library. It can detect various memory-related errors such as out-of-bounds access, user-after-free, user-after-return, use-after-scope, and

[9] https://clang.llvm.org/docs/AddressSanitizer.html

double-free, and invalid-free. Pointer Authentication (PA) [158] introduced in the ARMv8.3-A processor architecture protects memory-related vulnerabilities by computing and verifying *Pointer Authentication Codes* (PACs) for pointers through a set of instructions. The use of unverified pointers triggers memory translation faults.

**Performance evaluation using Address Sanitizer.** In this approach, we used four real-world applications (*nginx*, *httpd*, *lighttpd*, and *postgres*) and one benchmark program (*nbench*) to measure the impact of our prioritization on performance. We prepared three versions of each application/benchmark program: i) *Normal* version when no instrumentation is done, ii) *Asan* version when an application/benchmark program is instrumented with AddressSanitizer, and iii) *Asan+Prio.* version when an application/benchmark program is instrumented with AddressSanitizer instrumenting only the prioritized data objects. We ran each version of an application against benchmark workloads and measured the throughput. We measured the CPU Index for the *nbench* benchmark program. We used the wrk [10] workload generator for the web servers (i.e., *nginx*, *httpd*, and *lighttpd*) and sysbench [11] for postgres.

Table 5.4: Performance improvement of our prioritization technique over Address Sanitizer

| | | Instrumentation criteria | | | Overhead | |
|---|---|---|---|---|---|---|
| **Application** | **Performance metric** | **No Instr.** | **Instr. w/ asan** | **Instr. w/ asan+prio.** | **w/ asan** | **w/ asan +prio.** |
| nginx | throughput | 21957 | 12582 | 16225 | 42.7% | 26.1% |
| httpd | throughput | 47019 | 26030 | 34729 | 44.6% | 26.1% |
| lighttpd | throughput | 51552 | 29380 | 35684 | 43.0% | 30.8% |
| postgres | throughput | 1272 | 672 | 933 | 47.2% | 26.7% |
| nbench | CPU Index for 10 nbench algorithms | 810.43 | 501.01 | 677.98 | 38.2% | 16.3% |
| | | | | | **43.14%** | **25.2%** |

Table 5.4 shows the performance of the four applications and one benchmark in the three instrumentation criteria. We found that Address Sanitizer incurs around 43% overhead on average compared to the normal instrumentation criteria. When we incorporated our prioritization technique with Address Sanitizer, we observed the overhead is around 25% on average compared to normal,

[10] https://github.com/wg/wrk
[11] https://github.com/akopytov/sysbench

which improves performance by 18% and reduces the overhead by 42% compared to AddressSanitizer.

Figure 5.8 shows the performance comparison between the baseline, vanilla Address Sanitizer, and Address Sanitizer with prioritized data objects in the nbench byte benchmark dataset. On average, our approach incurs only a 12% slowdown whereas the vanilla address sanitizer incurs a 33% slowdown. We are continuing this experiment with the CPU Spec benchmark dataset.

**Performance evaluation using ARM PA.** We also evaluated the performance improvement of our prioritization technique using the ARM Pointer Authentication (PA) mechanism. We used ARM PA to sign and authenticate Store/Load IR instructions of our prioritized data objects/pointers in a program. We also signed and authenticated Load/Store IR instructions for all data objects/pointers in the program to compare with our technique. We used five real-world applications (*httpd-2.4.49*, *orzhttpd-0.0.6*, *sudo-1.8.3*, *lighttpd-1.4.17*, and *proftpd-1.3.0*) for this evaluation. The ideal approach for performance measurement is to run benchmarks against these programs (as we did in the Address Sanitizer case). Unfortunately, we could not build these programs and run benchmarks against them as the programs do not have support for cross-compilation which is necessary for compiling and building the programs for the ARM platform. However, we indirectly measure the performance improvement by computing the instrumentation size and number of Store/Load IR instructions to sign/authenticate.

Table 5.5: Reduction of instrumentation size and number of Loads/Stores when applying ARM PA considering prioritized data/object pointers compared to all data/object pointers.

| **Program** | **Reduction (%) when k% of prioritized objects/pointers** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Instrumentation size** | | | **Number of Loads** | | | **Number of Stores** | | |
| | k=10 | k=20 | k=100 | k=10 | k=20 | All | k=10 | k=20 | k=100 |
| httpd | 90 | 85 | 77 | 90 | 84 | 77 | 95 | 91 | 80 |
| orzhttpd | 96 | 96 | 92 | 99 | 99 | 97 | 100 | 100 | 95 |
| sudo | 59 | 54 | 29 | 54 | 49 | 29 | 97 | 92 | 51 |
| lighttpd | 32 | 31 | 21 | 25 | 25 | 18 | 91 | 85 | 50 |
| proftpd | 31 | 28 | 17 | 13 | 11 | 7 | 98 | 88 | 52 |
| **Average →** | **62%** | **59%** | **47%** | **56%** | **54%** | **46%** | **96%** | **91%** | **66%** |

Table 5.5 shows the reduction of instrumentation size and the number of Load/Store IR instructions

when we apply ARM PA considering different percentages (10%, 20%, and 100%) of our prioritized data objects/pointers compared to all data objects/pointers. On average, we can reduce the instrumentation size by around 62% by leaving around 56% and 96% Load and Store instructions uninstrumented, respectively without compromising security as we need to protect only around 11% of all prioritized data objects (refer to Table 5.3). Besides, we can reduce the instrumentation size and number of Load instructions nearly by 50% and the number of Store instructions by 66% in at the worst case (i.e., k=100%). These results suggest the promise of tuning security vs. performance according to an application's necessity.

## 5.5 Discussion

Our prioritization approach is different from the Clang Static Analyzer [12] as we prioritize data objects/pointers according to their sensitivity for vulnerabilities. While Clang Static Analyzer only warns for potential bugs, our goal is to make our prioritization approach generic for underlying instrumentation techniques or defenses (ARM PA [158], AddressSanitizer [175], MPX [104], etc.). We believe this makes our prioritization approach different from Clang Static Analyzer.

Another benefit of the prioritization technique is the ability to fine-tune between performance and security. We have demonstrated the minimum number of data objects (on average) that we need to protect for blocking exploits. However, one can tune this number by prioritizing security over performance or vice-versa for his/her application. This ability of fine-tuning performance and security adds the flexibility to keep an application protected while achieving demanded performance.

[12] `https://clang.llvm.org/docs/ClangStaticAnalyzer.html`

# Chapter 6

# Guidelines and Practical Considerations

The measurement results in Chapter 3 and Chapter 5 along with our systemization work in Chapter 4 taught us several lessons. However, to apply the lessons effectively and tackle real-world challenges, we need to make a set of guidelines and practical considerations. Next, we discuss these guidelines and practical considerations.

**Do not underestimate attackers' capabilities.** Attacks in system security domains are evolving fast. Besides, unique attackers' strategies make the anticipation of future attacks difficult. These unique strategies also make the generalization of attackers' capabilities challenging. In our work, we attempted to generalize and measure attackers' capabilities using various gadget sets such as the Turing-complete gadget set. Our approach of measuring attackers' capabilities through various gadget sets represents our best efforts, by no means the only way. For example, a pair of load and store gadgets may potentially replace a move-register gadget. This replacement relaxes the need for having a move-register gadget in the Turing-complete gadget set, though the replacement may not be directly equivalent due to possibly mismatching memory offsets of extended footprint load gadgets or the scarcity of minimum footprint load gadgets. Excluding load-n-store from the Turing-complete gadget set might underestimate attackers' capabilities, while including them might overestimate attackers' capabilities. This is why it is important to strike a balance between attackers' capabilities through our measurement. We attempted to balance the attackers' capabilities through our measurements by breaking down the Turing-complete gadget set into two smaller gadget sets: *i)* priority gadget set and *ii)* MOV TC gadget set. However, these smaller sets underestimate attackers' capabilities as some attackers may only need a few gadgets to perform their

attacks. Our measurements through these gadget sets suggest a way to assess generalized attackers' capabilities, but may not reflect too specific or very unique attackers' capabilities. Similarly, the upper bound measurement also does not guarantee protection against some attackers. A shorter interval may still allow attackers to gain information. We call the upper bound of re-randomization intervals as the "best-case" re-randomization interval from a defender's perspective because the defender has to re-randomize by the time of the interval, if not sooner. This raises the question regarding the effectiveness of "best-case" intervals over "worst-case" intervals. The "worst-case" interval indicates the time required to build a useful gadget chain using a minimal set of gadgets. In reality, attackers' goals vary. It is difficult to determine a minimum set of gadgets common and necessary across all attack chains.

**Do not overestimate attackers' capabilities.** Existing attack demonstrations assume the capability of a leaked code pointer for code reuse attacks [18, 27, 28, 185] and the capability of arbitrary read/write for data-oriented attacks [100, 101, 106, 133, 162, 168, 196, 219] through remote exploitation of a vulnerability. The leaked content or arbitrary read/capability is a must-have in launching exploits, however, most proof-of-concept attacks assume these capabilities. In reality, attackers may need to use memory disclosure vulnerabilities (e.g., heap overflows, use-after-free, type confusion, etc.) and weaknesses in system internals (e.g., vulnerabilities in the Glibc malloc implementation or its variants [9, 96], Heap Feng Shui [190], and Flip Feng Shui [160]) to leak memory contents or gain arbitrary read/write access. We seriously need to assess the practicality of this assumption. Besides, it is also necessary to evaluate how useful arbitrary reads are. Our assessment of examining the overall risk associated with stack, heap, and data segments can be the first step for this evaluation. However, more system measurement approaches are necessary to complete this assessment.

**Choose the right metric.** In our measurements, we have used the gadget availability and gadget quality (i.e., gadget corruption rate) metrics. However, are these the right metrics to use? For example, the measurement using gadget availability has a limitation of precisely estimating attackers' capabilities in all scenarios. Thus, refining the gadget metrics would be useful. Similarly,

we need to further explore the benefits of gadget corruption rate as a metric. We can design new randomization approaches aiming at increasing the register corruption rates through register-level heuristics in gadgets and overall gadget corruption rates, e.g., through guided and strategically permuting registers that maximize the gadget corruption rate without compromising the correctness of normal executions. Designing randomization solutions to increase the register corruption rate in gadgets would be interesting as a high register corruption rate would make attacks unreliable.

**Consider wider applicability.** Our measurements in this dissertation specifically target the C/C++ programs in the x86 platform. However, such measurements must be performed for other domains (e.g., non-C/C++ languages) as well as other platforms such as ARM or embedded systems. While the data pointer protection is applicable for other platforms, it may not be applicable for non-C/C++ languages such as Python or Java. Similarly, the gadget availability metric can work with other platforms given the construction of gadgets needs to be adjusted based on the underlying Instruction Set Architectures.

**Identify when precision matters over speed or vice-versa.** The key of any data flow analysis is the points-to analysis. The precision of points-to analysis makes the data flow analysis accurate and precise. However, the precision of a points-to analysis comes with a cost of speed. In our data object/pointer prioritization work in Chapter 5, we have used the Anderson points-to analysis [7], which is very precise but can be impractical for complex and large applications, especially applications having usage of a large number of function pointers. In this case, a less precise but faster points-to analysis technique such as Steensgaard's algorithm [191] can be useful if the precision is not critical. For example, our one-time static analysis-based prioritization can tolerate some speed issue, which enables our prioritization to use precise points-to analysis such as Anderson algorithm [7], or even more precise approaches such as flow-sensitive or field-sensitive points-to analysis.

# Chapter 7

# Conclusion and Future Work

In our **first work (chapter 3)**, we presented multiple general methodologies for quantitatively measuring the ASLR security under the JIT-ROP threat model and conducted a comprehensive measurement study. One method is for computing the number of various types of gadgets and their quality. Another method is for experimentally determining the upper bound of re-randomization intervals. The upper bound helps guide re-randomization adopters to make more informed configuration decisions. Our experiments showed that fine-grained code randomization up to basic block level does not substantially weaken attackers' capabilities, however, instruction-level does. The primary reason is that function, basic-block, or machine register level fine-grained randomization preserves Turing completeness, however, instruction-level randomization does not. We also found that a stack has a higher risk of being the source of memory leak than a heap or data segment. The reduction in the gadget availability by fine-grained randomization does not substantially weaken attackers' capabilities, as attackers only need one gadget per type. Although in some cases, the number of minimum footprint gadgets are reduced to zero, there are still plenty of extended footprint gadgets available. We observed that the register corruption rate is slightly higher ($\sim$6%) under fine-grained randomization. As one of the few measurement papers in the ROP space, the significance of our work is beyond these security findings. Our metrics, methodologies, and synthesized knowledge about advanced exploits would be useful beyond this specific study. The insights gained from our experimental results enable us to outline several promising new attack and defense research directions.

A few open problems stand out from our experiments in our work in chapter 3: *i)* How easy is it

for attackers to obtain a leaked pointer? How realistic is it to assume a leaked pointer is known, as is done in most attack demonstrations? Systematically measuring and quantifying the pointer leakage risks in applications would be useful. *ii)* How to develop robust fine-grained ASLR tools that can handle complex libraries such as Glibc? Currently, there is no known open source solution that reliably provides this capability in Linux.

Traditionally, both coarse-grained (e.g., PaX ASLR [199]) and fine-grained (e.g., SR [47], CCR [111], Remix [38], Binary stirring [213], ILR [97] and ASLP [108]) randomizations use entropy to measure the effectiveness of hindering code-reuse attacks. However, such an entropy measure is not useful under the JIT-ROP threat model, as chunks of code are still available. Inclusion of distances between permuted functions or basic blocks for computing entropy would not work either, because the code's semantic connectivity (e.g., through *call* and *jmp*) is still not captured. Code connectivity is what JIT-ROP attacks leverage to discover code pages. In comparison, our measurement methodology more accurately reflects JIT-ROP capabilities and is more meaningful under the JIT-ROP model. How to design an entropy-like metric to capture the degree of code isolation or the **semantic connectivity** in code is an interesting open problem.

Our work in chapter 3 has several limitations. For example, both CFI and XoM defenses are powerful and have capabilities to prevent JIT-ROP attacks. These two defenses with continuous re-randomization would be even more powerful. However, we did not enforce CFI and XoM in this work to isolate an individual defense's security impact. In this work, we addressed many important questions related to fine-grained (re-)randomization, not yet answered by the literature. We leave the analysis and measurement of CFI and XoM as future research.

Our current work does not measure zombie gadgets [186] and microgadgets [99]. The gadgets that are available after applying destructive read defenses (e.g., XnR [11], NEAR [215], Readactor [52], and Heisenbyte [197]) are called *zombie gadgets* [186]. Destructive read defenses only allow code execution, no read after execution. In this way, destructive reads can limit gadget availability, but cannot eliminate all gadgets. We plan to assess the availability of zombie and microgadgets in our

future work.

Another limitation is that we assume the code pointer obfuscation is not enforced. If enforced, code pointer obfuscation (e.g., CPI [114, 115], Oxymoron [12]) could make JIT-ROP code page discovery less effective, reducing the gadget availability. Understanding how code pointer obfuscation impacts JIT-ROP and measuring the effectiveness of this defense under various attack conditions (e.g., Isomeron [62] and COOP [169]) are interesting problems.

One limitation of time-based re-randomization schemes is that the intervals need recalculation with the evolution of hardware or a program itself. Event-based re-randomization schemes can be effective in this case. However, event-based schemes may trigger unnecessary re-randomization if events are frequent, e.g., re-randomizing every time a program outputs [16].

In our **second work in chapter 4**, we systematized the current knowledge of data-oriented exploits and applicable defense mechanisms. We hope that this systematization will stimulate a broader discussion about possible ways to defend against data-oriented attacks. We highlight some interesting future directions in this area.

*Automation of Small Footprint DOP Attacks.* An interesting research direction is how to minimize the footprints (*i.e.*, side effects) of a DOP attack while achieving the same attack goal. Attackers may prefer data-oriented gadgets that cause a minimum deviation from normal executions. Such a selection process requires automation to be efficient. Besides automation, one also needs to define metrics to measure the footprints, *i.e.*, the amount of alteration caused by a DOP execution. Ispoglou *et al.* [105] made the first step towards automating data-oriented programming through a powerful Block Oriented Programming Compiler (BOPC). Searching for gadget chains under specific constraints is an interesting research direction.

*Assessment of Programs' Susceptibility to Data-Oriented Attacks.* Such a characterization – statically or dynamically – would help one understand the threats that CFI cannot protect against. A promising direction is to quantify the degree of control-flow decisions that are dependent on adversarially controlled data (*e.g.*, user input). Such a characterization also helps prioritize the defense

effort, enabling one to address programs with the highest susceptibility first.

*Deep Learning for Control-Flow Behavior Modeling.* Non-control data violations may have impacts on control-flows in different locations with long distances in a program. How to detect incompatible control-flow paths, given a relatively long control-flow sequence, is challenging. Deep learning techniques have shown promises in detecting anomalies in different applications. An interesting research direction is to apply deep learning algorithms to model program behaviors for anomaly detection. For example, Recurrent Neural Network (RNNs), especially Long Short-Term Memory (LSTM) models, can be leveraged to capture temporal relations contained in univariate or even multivariate data. Such techniques may have the potential to detect incompatible control-flow paths given an extreme long control-flow sequence [127]. However, one challenge of deep learning based detection is the lack of labeled attack data given the difficulty to construct different DOP/BOP exploits. In addition, attackers may exploit adversarial machine learning techniques to evade detection by obfuscating control-flow behaviors under data-oriented attacks.

In our **third work in chapter 5**, we proposed a generic and adaptable data object/pointer prioritization framework that can be easily integrated with various data space protection countermeasure such as ARM PA [158], MPX [104], Softbound [134], AddressSanitizer [175], and Intel's CET [103]. The overall results suggest that the simple rule-based heuristics are simple but very powerful. Our exploit and vulnerability-driven rule-based heuristics give the flexibility to add new rules when necessary. Our experimental evaluations using 33 ground truths data objects/pointers from 18 programs including real-world server applications and 10 testcases from SAR dataset showed the successful identification of 32 ground truths with a 42% performance overhead reduction compared to AddressSanitizer. Our experimental results showed that the number of sensitive data objects in an application is as low as 3% for real-world applications. On average, we can filter out 82% of data objects that do not need protection.

Our approach may generate false positives in some conditions. For example, if a data object is well-bounded and even though the object is dependent on an external user. Since our heuristics

only check for the existence of bounded conditions, does not check right/wrong conditions, may falsely mark an object sensitive while the object is safe.

Our prioritization approach has some limitations. The first limitation is that our approach may miss some sensitive data objects/pointers as we discussed in Section 5.4.2. That means our approach cannot completely eliminate false negatives. One false negative in our approach was the inability to detect a target data object that is operationally dependent on another tainted data object or variable. There is no data-flow relationship between them. While this type of false negatives is challenging to detect, it opens new research directions. One way to detect this type of false positives is to taint a block of code that is dependent on another data variable. However, the size of the block of the code is important as the block of code needs to be very precisely dependent on the data variable. Otherwise, there is a chance of increasing false positives with the increase of the size of the code block.

Another limitation of this work is evaluating our work against launching end-end data-oriented exploits. However, we manually cross-check the detection/prioritization of data object(s) that several data-oriented exploits (refer to Table 5.2) utilize and found that all the target data objects are within the top 11% of our prioritized data objects, on average (Table 5.3).

# Bibliography


[1] The defense advanced research projects agency (darpa). https://github.com/CyberGrandChallenge/samples. Accessed Feb 12, 2020.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[4] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[5] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. Yao. Methodologies for quantifying (re-) randomization security and timing under jit-rop. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1803–1820, 2020.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[7] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[8] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[9] P. Argyroudis and C. Karamitas. Exploiting the jemalloc memory allocator: Owning firefoxs heap. *Blackhat USA*, 2012.

[10] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *NDSS*, 2015.

[11] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[12] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, pages 433–447, 2014.

[13] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 246–251. IEEE, 2007.

[14] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, et al. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 337–358. Springer, 2018.

[15] S. Bhatkar and R. Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[16] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[17] A. Biondo, M. Conti, and D. Lain. Back to the epilogue: Evading control flow guard via unaligned targets. NDSS, 2018.

[18] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[19] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[20] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[21] M. D. Brown and S. Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

[22] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[23] T. H. Bug. http://heartbleed.com, 2020. Accessed April 03, 2020.

[24] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[25] E. D. by Offensive Security. Ht editor 2.0.20 - local buffer overflow (rop). `https://www.exploit-db.com/exploits/22683`, 2012. Last accessed 05 May 2020.

[26] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008, 2008.

[27] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On

the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[28] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, pages 385–399, 2014.

[29] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[31] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[32] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*, pages 163–177. Springer, 2009.

[33] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29, 2011.

[34] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 167–178, 2017.

[35] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[36] X. Chen, H. Bos, and C. Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 514–529. IEEE, 2017.

[37] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[38] Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.

[39] L. Cheng, S. Ahmed, H. Liljestrand, T. Nyman, H. Cai, T. Jaeger, N. Asokan, and D. Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[40] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. D. Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Secure Development, SecDev 2019*, pages 114–128. Institute of Electrical and Electronics Engineers Inc., 2019.

[41] L. Cheng, K. Tian, and D. Yao. Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[42] L. Cheng, K. Tian, D. Yao, L. Sha, and R. A. Beyah. Checking is believing: event-aware program anomaly detection in cyber-physical systems. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[43] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attack. In *NDSS*, 2014.

[44] T.-c. Chiueh and F.-H. Hsu. Rad: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417. IEEE, 2001.

[45] M. Cole and A. Prakash. Simplex: Repurposing intel memory protection extensions for information hiding, 2020.

[46] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963. ACM, 2015.

[47] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016(4):454–469, 2016.

[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[49] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[50] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[51] S. C. Cowan, S. R. Arnold, S. M. Beattie, and P. M. Wagle. Pointguard: method and system for protecting programs against pointer corruption attacks, July 6 2010. US Patent 7,752,459.

[52] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[53] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 763–780, 2015.

[54] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307. IEEE, 2014.

[55] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News*, 41(1):219–228, 2013.

[56] Cyclone. http://cyclone.thelanguage.org/, 2002. [Accessed 08-12-2019].

[57] Daniel Moghimi. Subverting without EIP. `https://moghimi.org/blog/subverting-without-eip.html`, 2014. Last accessed 6 January 2021.

[58] R. V. . E. Database. Firebird relational database cnct group number buffer overflow. `https://www.rapid7.com/db/modules/exploit/windows/misc/fb_cnct_group`, 2018. Last accessed 05 May 2020.

[59] R. V. . E. Database. Proftpd 1.3.2rc3 - 1.3.3b telnet iac buffer overflow (linux). `https://www.rapid7.com/db/modules/exploit/linux/ftp/proftp_telnet_iac`, 2018. Last accessed 05 May 2020.

[60] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Xifer: A software diversity tool against code-reuse attacks. In *4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012)*, volume 174. Citeseer, 2012.

[61] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[62] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[63] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.

[64] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[65] DEP. Data execution prevention. `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`, 2018. Last accessed 09 May 2020.

[66] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[67] S. Dolan. mov is turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[68] G. J. Duck and R. H. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[69] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, 2017.

[70] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[71] H. Etoh. Gcc extension for protecting applications from stack-smashing attacks. *http://www.trl. ibm. com/projects/security/ssp*, 2000.

[72] H. Etoh. Gcc extension for protecting applications from stack-smashing attacks (propolice), 2003.

[73] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, 2015.

[74] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[75] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 28–39. ACM, 2018.

[76] T. I. for November. https://www.tiobe.com/tiobe-index/, 2020. Accessed November 30, 2020.

[77] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 15–26, 2008.

[78] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26, 2009.

[79] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*, volume 112, 2001.

[80] I. Fratrić. Ropguard: Runtime prevention of return-oriented programming attacks. In *Technical report*, 2012.

[81] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[82] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[83] M. Ghaffarinia and K. W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1009–1022, 2019.

[84] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[85] W. Glozer. wrk-a http benchmarking tool. `https://github.com/wg/wrk`, 2018. Last accessed 03 May 2020.

[86] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[87] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, 2014.

[88] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and

H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119, 2016.

[89] E. Göktaş, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.

[90] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 13, 2017.

[91] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[92] A. Gupta, J. Habibi, M. S. Kirkpatrick, and E. Bertino. Marlin: Mitigating code reuse attacks using code randomization. *IEEE Transactions on Dependable and Secure Computing*, 12(3):326–337, 2014.

[93] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[94] Hardware-assisted AddressSanitizer". https://clang.llvm.org/docs/ HardwareAssistedAddressSanitizerDesign.html, 2017. [Online; accessed 03-31-2019].

[95] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566. IEEE, 2017.

[96] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. *arXiv preprint arXiv:1804.08470*, 2018.

[97] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[98] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE Computer Society, 2013.

[99] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 7–7. USENIX Association, 2012.

[100] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[101] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[102] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[103] Intel. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2019. Last accessed March 24, 2020.

[104] Introduction to Intelő Memory Protection Extensions. `https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html`, 2013. [Accessed 03-24-2020].

[105] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882. ACM, 2018.

[106] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. "the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 791–804, 2016.

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

[108] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[109] J. Kim and Y. I. Eom. Fast and space-efficient defense against jump-oriented programming attacks. In *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 7–10. IEEE, 2015.

[110] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[111] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

[112] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `https://users.suse.com/~krahmer/no-nx.pdf`, 2005. Last accessed 10 May 2020.

[113] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[114] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, volume 14, 2014.

[115] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *The Continuing Arms Race*, pages 81–116. Association for Computing Machinery and Morgan & Claypool, 2018.

[116] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732, 2013.

[117] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[118] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with" return-less" kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208, 2010.

[119] M. libc. A lightweight standard c library. `https://www.musl-libc.org`, 2011. Last accessed 09 May 2020.

[120] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *Acm SIGPLAN Notices*, 35(11):168–177, 2000.

[121] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 177–194, 2019.

[122] D. Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server, 2003.

[123] C. Liu, Z. Yang, Z. Blasingame, G. Torres, and J. Bruska. Detecting data exploits using low-level hardware information: A short time series approach. In *Proceedings of the First Workshop on Radical and Experiential Security*, pages 41–47. ACM, 2018.

[124] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[125] K. Lu, W. Lee, S. Nürnberger, and M. Backes. How to make aslr win the clone wars: Runtime re-randomization. In *NDSS*, 2016.

[126] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[127] Y. Luo, Y. Xiao, L. Cheng, G. Peng, and D. D. Yao. Deep learning-based anomaly detection in cyber-physical systems: Progress and opportunities, 2020.

[128] G. Maisuradze, M. Backes, and C. Rossow. What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In *USENIX Security Symposium*, pages 139–156, 2016.

[129] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[130] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.

[131] J.-W. Min, S.-M. Jung, D.-Y. Lee, and T.-M. Chung. Jump oriented programming on windows platform (on the x86). In *International Conference on Computational Science and Its Applications*, pages 376–390. Springer, 2012.

[132] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[133] M. Morton, J. Werner, P. Kintis, K. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182. IEEE, 2018.

[134] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[135] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[136] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[137] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[138] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN notices*, 42(6):89–100, 2007.

[139] T. Newsham. Non-exec stack. *Bugtraq mailing list*, 2000.

[140] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[141] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[142] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi. Hardscope: Hardening embedded systems against data-oriented attacks. In *56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[143] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi. Hardscope: Hardening embedded systems against data-oriented attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[144] N. I. of Standards and Technology. Software assurance reference dataset. `https://samate.nist.gov/SRD/testsuite.php`. Last accessed 16 December 2021.

[145] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *SIGMETRICS Perform. Eval. Rev.*, 46(1):111112, June 2018.

[146] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[147] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[148] T. Palit, F. Monrose, and M. Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 598–611, 2019.

[149] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.

[150] V. Pappas, M. Polychronakis, and A. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, pages 447–462, 2013.

[151] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[152] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[153] A. Peslyak. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[154] J. Pewny, P. Koppe, and T. Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 111–126, 2019.

[155] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[156] M. Prasad and T.-c. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[157] S. Priyadarshan, H. Nguyen, and R. Sekar. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference*, pages 401–414, 2020.

[158] Qualcomm Technologies Inc. Pointer Authentication on ARMv8.3. `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`, 2017.

[159] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz. Codarr: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 494–505, 2020.

[160] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, 2016.

[161] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[162] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 366–381. IEEE, 2017.

[163] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, et al. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *NDSS*, 2017.

[164] A. Sadeghi, F. Aminmansour, and H. R. Shahriari. Tiny jump-oriented programming attack (a class of code reuse attacks). In *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 52–57. IEEE, 2015.

[165] A. Sadeghi, S. Niksefat, and M. Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14(2):139–156, 2018.

[166] SAFE secure computing platform. http://www.crash-safe.org/, 2019. [Accessed 08-12-2019].

[167] S. Schirra. Ropper tool. `https://github.com/sashs/Ropper`, 2014. Last accessed 4 July 2018.

[168] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[169] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[170] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *International Workshop on Recent Advances in Intrusion Detection*, pages 88–108. Springer, 2014.

[171] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.

[172] E. J. Schwartz, C. F. Cohen, J. S. Gennari, and S. M. Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1801, 2020.

[173] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 54–65, 2014.

[174] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[175] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[176] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[177] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[178] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[179] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[180] Shellphish. Educational Heap Exploitation: how2heap . `https://github.com/shellphish/how2heap`, 2019. Last accessed 6 January 2021.

[181] Y. Shoshitaishvili, C. Kruegel, G. Vigna, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[182] X. Shu, D. Yao, N. Ramakrishnan, and T. Jaeger. Long-span program behavior modeling and attack detection. *ACM Transactions on Privacy and Security (TOPS)*, 20(4):1–28, 2017.

[183] K. Sinha and S. Sethumadhavan. Practical memory safety with rest. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 600–611. IEEE, 2018.

[184] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[185] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[186] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 954–968. IEEE, 2016.

[187] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[188] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[189] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[190] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[191] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[192] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[193] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[194] Z. Sun, B. Feng, L. Lu, and S. Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[195] SVF-tools. Static Value Flow (SVF). `https://github.com/svf-tools/SVF/wiki/Technical-documentation`, 2021. Last accessed September 6, 2021.

[196] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[197] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 256–267. ACM, 2015.

[198] C. Team. Universal dep/aslr bypass with msvcr71.dll and mona.py. `https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/`, 2018. Last accessed 10 February 2018.

[199] P. Team. Pax address space layout randomization (aslr). 2003.

[200] The Rust Programming Language. https://www.rust-lang.org/, 2019. [Accessed 08-12-2019].

[201] G. Torres and C. Liu. Can data-only exploits be detected at runtime using hardware events?: A case study of the heartbleed vulnerability. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 2. ACM, 2016.

[202] S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017*, pages 1–14, 2017.

[203] Uptrends. Website speed test. `https://www.uptrends.com/tools/website-speed-test`, 2020. Last accessed 03 May 2020.

[204] A. van de Ven and I. Molnar. Exec shield. `https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004. Last accessed 26 September 2018.

[205] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2017.

[206] V. Van Der Veen, E. Göktaş, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[207] J. Vanegue. The automated exploitation grand challenge. `https://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf`, 2013. Last accessed Feb 12, 2020.

[208] S. S. Vendicator. A stack smashing technique protection tool for linux. *World Wide Web, http://www. angelfire. com/sk/stackshield/info. html*, 2000.

[209] J. A. Vidrine, N. C. Allegra, S. P. Cooper, and G. D. Hughes. Fine-grained address space layout randomization, Mar. 31 2016. US Patent App. 14/503,212.

[210] K. Volodymyr, S. Laszlo, P. Mathias, C. George, and R. Sekar. Code-pointer integrity. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[211] K. vz'One Enchant. ftpbench-benchmark for load testing ftp servers. `https://github.com/selectel/ftpbench`, 2014. Last accessed 03 May 2020.

[212] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[213] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 157–168. ACM, 2012.

[214] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[215] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 35–46. ACM, 2016.

[216] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, pages 367–382, 2016.

[217] R. Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[218] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[219] J. Xiao, H. Huang, and H. Wang. Kernel data attack is a realistic security threat. In *Inter-*

*national Conference on Security and Privacy in Communication Systems*, pages 135–154. Springer, 2015.

[220] D. Yao, X. Shu, L. Cheng, and S. J. Stolfo. Anomaly detection as a service: challenges, advances, and opportunities. *Synthesis Lectures on Information Security, Privacy, and Trust*, 9(3):1–173, 2017.

[221] S. H. Yong and S. Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 307–316, 2003.

[222] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[223] H. Zhang, D. D. Yao, N. Ramakrishnan, and Z. Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security*, 58:180–198, 2016.

[224] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, pages 337–352, 2013.