UNIVERSITY OF CALIFORNIA,
IRVINE

Practical Run-Time Mitigations Against Data-Oriented Attacks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Prabhu karthikeyan Rajasekaran

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ardalan Amiri Sani
Professor Anton Burtsev

2020

Portions of Chapter 1, 4, and 5 © 2020 ACM.
Reprinted, with permission, from **CoDaRR: Continuous Data Space Randomization against Data-Only Attacks**, Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz., in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, **ASIA CCS 2020**.

Portions of Chapter 1, 2, and 5 © 2019 IEEE.
Reprinted, with permission, from **SoK: Sanitizing for Security**, Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Per Larsen, Stijn Volckaert, and Michael Franz., in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, **IEEE S&P 2019**.

Portions of Chapter 1, 3 and 5 © 2017 USENIX.
Reprinted, with permission, from **Venerable Variadic Vulnerabilities Vanquished**, Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer, in *Proceedings of the 2017 USENIX Annual Technical Conference*, **USENIX Security 2017**.

All other materials © 2020 Prabhu karthikeyan Rajasekaran

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

VITA **viii**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**
1.1 Control-flow Attacks and Defenses 1
1.2 Data-oriented Attacks and Defenses 2
1.3 Contributions 4
1.3.1 Variadic Function Sanitizer and Mitigation 4
1.3.2 Dynamic Data Space Randomization 4
1.3.3 Summary of Contributions 5

2 Background **7**
2.1 Data-Oriented Attacks 8
2.1.1 Direct Data Manipulation 8
2.1.2 Data Oriented Programming 9
2.2 Memory Corruption Vulnerabilities 11
2.2.1 Memory Safety Violations 11
2.2.2 Use of Uninitialized Variables 13
2.2.3 Pointer Type Errors 13
2.2.4 Variadic Function Misuse 15
2.2.5 Other Vulnerabilities 16
2.3 Side-Channel Attacks 17
2.3.1 Transient Execution Attacks 17

3 A comprehensive approach to fixing variadic vulnerabilities **20**
3.1 Motivation 21
3.2 Background 24
3.2.1 Variadic functions 24

3.2.2 Variadic functions ABI 25
3.2.3 Variadic attack surface 26
3.2.4 Format string exploits 27
3.3 Threat model 28
3.4 Design 29
3.4.1 Analysis and Instrumentation 29
3.4.2 Runtime support 30
3.4.3 Challenges and Discussion 32
3.5 Implementation 36
3.6 Evaluation 39
3.6.1 Case study: CFI effectiveness 40
3.6.2 Exploit Detection 43
3.6.3 Prevalence of variadic functions 45
3.6.4 Firefox 48
3.6.5 SPEC CPU2006 49
3.6.6 Micro-benchmarks 49

4 Moving Beyond Data Space Randomization **51**
4.1 Motivation 51
4.2 Background 54
4.3 Attacking DSR 57
4.3.1 Threat Model 58
4.3.2 Attack 1 – Direct Memory Disclosure 59
4.3.3 Attack 2 – Transient Execution 60
4.4 Design of a countermeasure 63
4.4.1 Challenges 63
4.4.2 Overview 65
4.4.3 Compile-time Analysis and Instrumentation 67
4.4.4 Mask Table and Tracking Live Masks 68
4.4.5 Value Mask Mappings 69
4.4.6 Run-time Monitor 70
4.5 Details of our proof-of-concept implementation 72
4.5.1 Code Pointer Fixups 73
4.5.2 Optimizations 75
4.6 Evaluation 75
4.6.1 Performance 75
4.6.2 Security Analysis 80
4.7 Discussion 82
4.7.1 Generating Variants 82
4.7.2 Attack Against Heap Metadata 82

5 Related Work **84**
5.1 Dynamic Bug Finding – Sanitizers 84
5.2 Variadic Function Attack Surface Mitigations 86
5.3 Security Mitigations 87

5.3.1 Control-Flow Exploit Defenses 88
5.3.2 Data-Oriented Exploit Defenses 90
5.4 Rerandomization Defenses 92

6 Conclusions **94**

Bibliography **96**

LIST OF FIGURES

Page

3.1 Overview of our compilation pipeline. Our instrumentation passes run right after the C/C++frontend, while our runtime library, `hexvasan.a`, is merged into the final executable at link time. 30
3.2 Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance. 50

4.1 Direct memory disclosure attack on a reused XOR key. $\mathbb{R}$: read V, unmasking with key K_R, $\mathbb{W}$: write V, masking with key K_W. 59
4.2 Overview of CoDaRR's main components. 63
4.3 CoDaRR run-time rerandomization process. 67
4.4 Average throughput of Nginx when protected by CoDaRR. We plot the median transfer rate for 5M requests against different rerandomization intervals going from 500ms to 10s. The green line shows throughput for baseline DSR without rerandomization. 77
4.5 Average throughput of Thttpd using the same configuration as in Figure 4.4. 78
4.6 SPEC 2006 performance of CoDaRR instrumentation (without rerandomization). Each column shows the median run time of the benchmark with CoDaRR, normalized to the median run time without our tool. 79

LIST OF TABLES

Page

2.1 Simulating MINDOP Data Oriented Programming Gadgets in C 10

3.1 Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection. 41
3.2 Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect); % shows the percentage of variadic call sites. The fifth and sixth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions. 45
3.3 Statistics of Variadic Functions for SPEC 2006 Benchmarks. 46
3.4 Performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better. 48
3.5 Performance overhead in micro-benchmarks. 50

4.1 Total number of static equivalence classes and number of allocations per class. 76
4.2 Average number of bytes rewritten at dynamic rerandomization time. Globals and Heap columns show bytes re-encrypted; Registers and Stack Spills are in-flight masks replaced. 76

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Michael Franz, for giving me an opportunity to conduct academic research in his research group, *Secure Systems and Software Lab (SSLLab)* . Thank you, Michael, for your unwavering support, mentorship, and guidance through the ups and downs of my professional and personal life during my time here at UCI. I am indebted to our lab alumnus Dr. Stephen Crane for taking me under his wing. Thank you, Stephen, for those countless hours of help in debugging our code and for setting lofty standards as a computer scientist and as a human being that I strive to emulate.

The postdocs at SSLLab, both current and previous, I had the good fortune to work with have had a great impact on my growth as a researcher and systems engineer. I am deeply grateful to Dr. Per Larsen, Dr. Stijn Volckaert, Dr. Yeoul Na, Dr. David Gens, and Dr. Adrian Dabrowski for their advice, ideas, and research contributions. To the illustrious lab alumni Dr. Brian Belleville, Dr. Julian Lettner, Dr. Andrei Homescu and Dr. Mohaned Qunaibit, I thank you for your mentorship, support, and for letting me walk into your offices anytime to seek advice.

To my fellow student researchers at the lab Paul, Nikhil, Joseph, Anil, Taemin, Alex, Dokyung, Mitchel, Matthew, Fabian, Min-Yih, and Chinmay, I thank you for inspiring me every day with your hard work, talent, and creativity. I thank the SSLLab community most of all for your camaraderie.

To my summer internship mentors Kiarash Ghadianipour (Amazon 2015), Dr. David Molnar (MSR 2016), Dr. Christos Gkantsidis (MSR 2017), Dr. David Tarditi (MSR 2018), Chris Leary (Google 2019) and Robert Hundt (Google 2019), I thank you for the opportunity to work with you and for the countless hours of mentorship which has had a tremendous impact in my growth as a researcher and an engineer.

I especially thank my committee members Professor Ardalan Amiri Sani, and Professor Anton Burtsev for their time, consideration, and constructive feedback.

This research is based on work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-16-C-0260, FA8750-15-C-0124 and FA8750-15-C-0085, the National Science Foundation under awards CNS-1513783, CNS-1657711, and CNS-1619211, the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, the Intel Corporation, as well as gifts from Oracle Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research (ONR) or its Contracting Agents, the National Science Foundation (NSF), or any other agency of the U.S. Government.

I would like to thank my friends Shreekanth, Vinay, Sudeeptha, Chloe, Holly, and my family for their love and support without which this journey would not have been possible.

VITA

Prabhu karthikeyan Rajasekaran

EDUCATION

Doctor of Philosophy in Computer Science **2020**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2016**
University of California, Irvine *Irvine, California*

EXPERIENCE

Graduate Research Assistant **2015–2020**
University of California, Irvine *Irvine, California*

Software Engineering Intern **Summer 2019**
Google LLC *Sunnyvale, California*

Research Intern **Summer 2018**
Microsoft Corporation *Redmond, Washington*

Research Intern **Summer 2017**
Microsoft Corporation *Cambridge, UK*

Research Intern **Summer 2016**
Microsoft Corporation *Redmond, Washington*

Software Engineering Intern **Summer 2015**
Amazon.com, Inc. *Seattle, Washington*

Senior Software Engineer **2012–2014**
Autodesk Inc. *Hyderabad, India*

Senior Development Engineer **2010–2012**
Development Engineer **2008–2010**
Pramati Technologies *Hyderabad, India*

TEACHING EXPERIENCE

Teaching Assistant, Programming Languages **2017**
University of California, Irvine *Irvine, CA*

Teaching Assistant, Senior Design Project **2015**
University of California, Irvine *Irvine, CA*

REFEREED CONFERENCE PUBLICATIONS

CoDaRR: Continuous Data Space Randomization against Data-Only Attacks **ASIA CCS 2020**
Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz.
ACM Asia Conference on Computer and Communications Security

BinRec: Dynamic Binary Lifting and Recompilation **EuroSys 2020**
Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos and Michael Franz
ACM European Conference on Computer Systems

SoK: Sanitizing for Security **IEEE S&P 2019**
Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Per Larsen, Stijn Volckaert, and Michael Franz
IEEE Symposium on Security and Privacy

Venerable Variadic Vulnerabilities Vanquished **USENIX Security 2017**
Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer
USENIX Security Symposium

ABSTRACT OF THE DISSERTATION

Practical Run-Time Mitigations Against Data-Oriented Attacks

By

Prabhu karthikeyan Rajasekaran

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael Franz, Chair

Data-oriented exploits are growing in popularity as defenders are closing attack vectors related to control flow exploits. Data Oriented Programming (DOP), a generalization of data-oriented attacks, stitches together data-oriented gadgets and can perform Turing-complete computations in memory. Besides, data-oriented exploits modify the behavior of the program without violating its control flow, and therefore cannot be stopped by ubiquitous control flow mitigations such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and Control Flow Integrity (CFI).

A unified solution cannot address all memory corruption errors as there are different classes of memory corruption errors and each of them requires careful considerations. The arguments to variadic functions in C and C++ cannot be statically type-checked by the compiler and therefore expose a significant memory corruption attack surface. In this dissertation, we present a comprehensive approach that effectively type-checks variadic function arguments and thus addresses the attacks that exploit variadic function related memory corruption errors. Our evaluation shows that our solution is practical as the measured overhead is negligible (0.45%) and effective as we show in several case studies.

To prevent data-oriented exploits from deterministically corrupting program data, a probabilistic fine-grained randomization defense Data Space Randomization (DSR) was proposed.

DSR randomizes the representation of program data in memory to prevent illegitimate usage of data. DSR employs static analysis to classify data objects into equivalence classes. DSR works by masking memory load and store instructions using XOR operations. Masks are chosen (i) to not interfere with intended data flows and (ii) such that masking likely interferes with unintended flows introduced by malicious program inputs.

We show using two new DSR bypass attacks, one that directly discloses memory and another using speculative execution, that the existing DSR defenses are still not secure. We then improve upon the state of the art DSR and propose the first dynamic DSR scheme resilient to disclosure attacks. Our mitigation continuously rerandomizes the masks used in loads and stores, and re-masks all memory objects to remain transparent w.r.t. program execution. Our evaluation confirms that our approach successfully thwarts these attacks with limited run-time overhead in standard benchmarks as well as real-world applications.

Chapter 1

Introduction

C and C++ remain the languages of choice for implementing low-level systems software such as operating system kernels, runtime libraries, and browsers since they are efficient and leave the programmer in full control of the underlying hardware. But that leaves the onus on the programmer to ensure that every memory access is valid and that no computation leads to undefined behavior. In practice, programmers routinely fall short of meeting these responsibilities and introduce bugs that make the software systems vulnerable to exploitation. Memory corruption errors are the fundamental building blocks upon which adversaries build their security exploits. The two broad categories of attacks that exploit memory corruption errors are i) control-flow attacks and ii) data-oriented attacks.

1.1 Control-flow Attacks and Defenses

Control-flow attacks corrupt instruction addresses stored in program memory such as return addresses or function pointers to divert the program's control flow through malicious code injection [54], code reuse [142], or return oriented programming (ROP) [133]. The goal of

adversaries here is to hijack the control flow of the program and execute arbitrary code of their choice.

Given their expressiveness, sophistication and severity, mitigating control-flow attacks has been a major focus of the security research community over the last decade. Data Execution Prevention [104] mitigation makes data memory non-executable, essentially eliminating traditional code injection attacks. Assumptions about the memory layout (location of return address in stack, function pointers in virtual table, etc.) is a key aspect in constructing control-flow attacks. Address Space Layout Randomization (ASLR) [143] is a probabilistic mitigation that randomizes the base addresses of stack, heap, code and global data sections to increase the failure rate of control-flow attacks. Control Flow Integrity (CFI) mitigation pre-calculates the set of valid targets as expressed in the source program for every indirect control flow transfer (iCFT) instruction. At run time, CFI checks if the observed target for each indirect control flow transfer is within the allowed target set.

1.2 Data-oriented Attacks and Defenses

Due to the widespread deployment of DEP, ASLR and CFI and the resulting increase in difficulty of launching control-flow exploits has forced the attackers to turn to *data-oriented attacks*. Data-oriented attacks, as opposed to control-flow attacks, corrupt the non-control data to manipulate the behavior of the program without violating the CFI enforcement policies. In their seminal work Chen et al. [28] showed that memory corruption exploits can manipulate the benign behavior of a program or leak sensitive data by corrupting non-control but security-critical data such as configuration data, user input data, user identity data, and decision-making data. Data-oriented attacks have grown in sophistication and have become an appealing alternative to control-flow attacks for system compromise [74, 108, 164, 134, 118, 55, 43, 29, 122]. Hu et al. [75] even proposed *Data-Oriented Programming* (DOP), a

systematic approach to building Turing-complete data-oriented attacks.

Control-flow mitigations can provide protection against some data-oriented attacks. For example, the ASLR randomizes the base offset addresses of data sections in program memory. But ASLR is a coarse-grained randomization solution which can be bypassed by multi-stage adaptive attacks that rely on information disclosure. Data-oriented attack defenses are comprehensive and generally protect against corruption of control and non-control data. Static Data Space Randomization(DSR) [15, 20, 14] is considered the strongest mitigation available against data-oriented attacks yet. DSR is a promising probabilistic exploit mitigation technique that randomizes the *representation* of data in memory to prevent deterministic corruption of data or plaintext information leakage. Compared to ASLR and compartmentalization [159, 7, 101] solutions, DSR can stop data-oriented attacks with higher probability. DSR applies static program analysis to partition all data accesses into disjoint *buckets* based on the aliasing relationships between variables. A pair of variables will be in the same bucket if and only if the analysis determines that legitimate data flows could exist between the two variables. At run time, the program then randomizes the *representation* of memory objects that fall into different buckets. This can be implemented efficiently by using lightweight `XOR` operations with a data-randomization mask (or "key") that is unique to each bucket.

DSR is only effective if there are many congruence buckets and the masks remain secret. DSR implementations embed masks in code sections to perform encryption and decryption operations to keep the masks from program memory (stack, heap and global sections). This design choice is made to protect against mask-leakage through information disclosure attacks. However, there are a number of possible ways to disclose masks, even if they cannot be read directly. For instance, an adversary could leverage memory vulnerabilities to disclose masked data. If the attacker can choose or inject data (*chosen-plaintext attack*, CPA), knows the data (*known-plaintext attack*, KPA), or guess them with high probability, he can infer the

value of a mask used for a bucket and all variables in it. Since all variables in a bucket have the same mask applied to them, this could then enable further leakage via a JIT-ROP style attack [145]. Alternatively, adversaries could exploit a remote side channel, such as *NetSpectre* [137], to directly disclose the secret key from the binary or the key cache.

1.3 Contributions

In this dissertation, we introduce the two following dynamic mitigation solutions.

1.3.1 Variadic Function Sanitizer and Mitigation

Several tools and defenses have been developed to contend with memory corruption exploits [152, 148, 89]. The current state of the art tools, however, overlook the memory corruption attack surface created by variadic functions (e.g. **printf**) in C and C++ programs. The *types* of the variadic arguments are dependent on the implementation of variadic functions and requires a stateful dynamic check to prevent variadic function exploits. In Chapter 3, we present HexVASAN, a dynamic bug finding and mitigation solution that addresses this attack vector.

1.3.2 Dynamic Data Space Randomization

Static Data Space Randomization protects against malicious access to data. In Chapter 4, we first demonstrate two novel attacks against DSR. Our attacks disclose DSR masks using direct memory disclosure or side-channel attacks. This allows us to write arbitrary plaintext data into the entire bucket whose mask we disclosed. Both of our attacks are sufficiently generic to work against any prior implementation of DSR, including those that rely on hardware

support to protect keys against disclosure, because they exploit design choices in the DSR scheme rather than implementation flaws.

We propose a new dynamic DSR scheme dubbed CoDaRR (Continuous Data Re-Randomization) which thwarts our presented and similar attacks by *dynamically* rerandomizing the masks and updating all program data accordingly.

1.3.3 Summary of Contributions

Specifically, we make the following contributions.

Chapter 3

- Design and implementation of a variadic function sanitizer HexVASAN on top of LLVM;
- A case study on large programs to show the prevalence of direct and indirect calls to variadic functions;
- Several exploit case studies and CFI bypasses using variadic functions.
- Evaluation that shows that our method is practically deployable as the measured overhead is negligible (0.45%) and effective as we show in several case studies.

Chapter 4

- We present two novel attacks against the current state of the art data defense DSR. Our attacks allow us to exploit attacker-controlled write operations to write arbitrary plaintext data to an encrypted memory location.
- We identify the key challenges towards building an additional layer of moving target defense on top of DSR which rerandomizes the DSR keys and data. We present a

new scheme which addresses these challenges. Our prototype, CoDaRR, supports on-demand as well as continuous rerandomization of data. Our evaluation shows that our rerandomization solution is resilient to the attacks against traditional DSR schemes.

- We evaluate the performance of implementation prototype using benchmark suites as well as applications such as *Nginx* and *Thttpd*. Our experiments show acceptable performance in most cases.

Chapter 2

Background

Data-oriented memory corruption exploits are growing in popularity due to the widespread adoption of control-flow exploit mitigation techniques. The first stage of a typical data-oriented exploit involves triggering a memory corruption error [32]. To familiarize the reader with data-oriented exploits, we dissect data-oriented exploit examples and discuss the structure of typical real world data-oriented exploits in Section 2.1. We then outline various types of memory corruption vulnerabilities which are the fundamental building blocks upon which the data-oriented exploits are built.

Current generation of randomization defenses, including the current state of the art mitigation available against data-oriented exploits – Data Space Randomization, are safe only as long as the randomization secrets remain indeed secret. Leaking such secrets (e.g. DSR encryption keys) can lead to adaptive multi-stage exploits that bypass these mitigations. Transient execution attacks have evolved in recent years and can assist in leaking randomization secrets of the programs protected by techniques such as DSR. In Section 2.3, we describe side-channel and transient execution attacks that we use in our DSR bypass attacks that we present in Chapter 4.

2.1 Data-Oriented Attacks

Data-oriented attacks manipulate the behavior of the program without violating CFI enforcement policies. Data-oriented attacks can be as simple as changing a single bit in program memory akin to the example shown in listing 2.1 or sophisticated multi-stage techniques; based on this insight Cheng et al. [32] classify data-oriented attacks into two categories 1) Direct Data Manipulation (DDM); and 2) Data Oriented Programming (DOP).

2.1.1 Direct Data Manipulation

In DDM exploits, the adversaries manipulate the target data directly to compromise the security of the system. The necessary prerequisite for DDM is that the attacker knows the exact memory location of the target non-control data. By analyzing the binary [70, 158, 130] or source code (in case of open sourced projects), or by reusing the runtime randomized address stored in the memory [74], the attackers can derive the target memory location. The memory corruption bugs outlined in Section 2.2 offer adversaries the attack surface necessary to build exploits that allow for arbitrary memory writes.

DDM attacks may corrupt one or more memory locations to achieve the intended malicious goal [108]. Listing 2.1 shows a simple example of a real world vulnerable SSH server code in which (non-control) decision-making data [28] is corrupted creating a DDM exploit.

Notice that the local variable *authenticated* declared in line 3 is used in deciding whether a remote user has passed the authentication check (line 17). The function call *packet_read* in line 6 invokes *detect_attack* function internally which has an integer overflow vulnerability (see Section 2.2.5). The attacker using this memory corruption vulnerability can simply overwrite the value of the security-critical variable *authenticated* to a non-zero value bypassing the remote authentication check.

```c
void do_authentication(char* user,...){
    ...
    int authenticated=0;
    ...
    while(!authenticated){
        type=packet_read(); //Corrupt authenticated
        /* Calls detect_attack() internally */
        switch(type){
            ...
            case SSH_CMSG_AUTH_PASSWORD:
                if(auth_password(user,password)) {
                    authenticated=1;
                    break;
                }
            case ...
        }
    if(authenticated) break;
    }

    do_authenticated(pw);
    /* Perform session preparation*/
}
```

Listing 2.1: Vulnerable SSH server

2.1.2 Data Oriented Programming

DDM is a prerequisite for DOP exploits. But DOP techniques are more sophisticated and expressive. The attackers, in DOP exploits, perform arbitrary computations in program memory [75] by chaining the execution of DOP gadgets. To build a successful DOP attack, the adversary must first identify DOP gadgets, then stitch them in an orderly fashion so that the Control Flow Integrity (CFI) policies are not violated while making sure that the data corruption goals are met.

Hu et al. [75] proposed *MINDOP* a simple Turing-complete mini-language with a virtual instruction set and virtual register operands to systematically construct DOP payloads. They introduced DOP gadgets which are instruction sequences in the vulnerable program that can mimic the operations defined by the MINDOP language. The MINDOP language (shown in

Semantics	C Instructions	DOP Gadgets
arithmetic/logical	a op b	*p op *q
assignment	a = b	*p = *q
load	a = *b	*p = **q
store	*a = b	**p = *q
jump	goto L	vpc = &input
Conditional jump	if a goto L	vpc = &input if *p

Table 2.1: Simulating MINDOP Data Oriented Programming Gadgets in C

Table 2.1) has 6 kinds of virtual instructions, each operating on virtual register operands. The first four virtual instructions are arithmetic/logical calculation, assignment, load and store operations. The last two are conditional and unconditional jumps which allow the implementation of *control structures* in a MINDOP virtual program. The control structure allows chaining of gadgets, and the x86 code sequences that simulate the virtual control operations are referred to as gadget dispatchers. Each virtual operation in MINDOP can be simulated by real x86 instruction sequences available in the vulnerable program and these sequences are common in real-world programs.

For example, Hu et al. [75] presented a DOP attack on ProFTPd server that leaks the OpenSSL private key stored in heap memory. This attack leverages an arbitrary memory write ability offered by a buffer overflow vulnerability (CVE-2006-5815) to i) build a data exfiltration channel from server to client, ii) replace data shown to the client with that of program's internal state (e.g. data pointers) and once the target private key's location was calculated with a series of interactions with the server, iii) leak the key once again using the exfiltration channel.

Ispoglou et al. [80] recently proposed Block-Oriented Programming (BOP) and introduced block-oriented programming compiler (BOPC) that helps further in automatic generation of DOP exploits. BOP follows the DOP exploit model but the BOP gadgets are entire basic blocks. BOPC given a target binary and DOP attack payload identifies DOP gadgets (basic

blocks) and stitches them together in a manner consistent with that of the legal control flow of the program.

The data-oriented attack examples and the techniques that are available to build these attacks outlined in this section demonstrates the severity, effectiveness and sophistication of data-oriented exploits and why they deserve careful attention from security researchers.

2.2 Memory Corruption Vulnerabilities

For performance reasons, C and C++ do not verify that a memory access is valid at run-time and invalid memory accesses are considered *undefined behavior*. Moreover, there is no unified solution that fits all memory corruption errors; we must contend with well-defined behaviors in C and C++ that are potentially dangerous due to the absence of type and memory safety. We briefly describe in this section the memory corruption bugs and how they can be exploited. The ProFTPd data-oriented attack referenced in the previous section [75] for example leveraged a combination of two memory corruption errors which offered the attackers with the ability to manipulate program memory.

2.2.1 Memory Safety Violations

A program is *memory safe* if pointers in the program only access their *intended referents*, while those intended referents are valid. The intended referent of a pointer is the object from whose base address the pointer was derived. Depending on the type of the referent, it is either valid between its allocation and deallocation (for heap-allocated referents), between a function call and its return (for stack-allocated referents), between the creation and the destruction of its associated thread (for thread-local referents), or indefinitely (for global referents).

```
struct A { char name[7]; bool isAdmin; };
struct A a; char buf[8];
memcpy(/* dst */ a.name, /* src */ buf, sizeof(buf));
```

Listing 2.2: Intra-object overflow vulnerability which can be exploited to overwrite security-critical non-control data

Memory safety violations are among the most severe security vulnerabilities and have been studied extensively in the literature [152, 157]. Their exploitation can lead to code injection [10], control-flow hijacking [147, 111, 142], privilege escalation [28], information leakage [151], and program crashes.

Spatial Safety Violations

Accessing memory that is not (entirely) within the bounds of the intended referent of a pointer constitutes a spatial safety violation. Buffer overflows are a typical example of a spatial safety violation. A buffer overflow happens when the program writes beyond the end of a buffer. If the intended referent of a vulnerable access is a subobject (e.g., a struct field), and if an attacker writes to another subobject within the same object, then we refer to this as an *intra-object overflow*.

Listing 2.2 shows an intra-object overflow vulnerability which can be exploited to perform a privilege escalation attack.

Temporal Safety Violations

A temporal safety violation occurs when the program accesses a referent that is no longer valid. When an object becomes invalid, which usually happens by explicitly deallocating it, all the pointers pointing to that object become *dangling*. Accessing an object through a dangling pointer is called a *use-after-free*. Accessing a local object outside of its scope or after

the function returns is referred to as *use-after-scope* and *use-after-return*, respectively. This type of bug becomes exploitable when the attacker can reuse and control the freed region, as illustrated in Listing 2.3.

```
1    struct A { void (*func)(void); };
2    struct A *p = (struct A *)malloc(sizeof(struct A));
3    free(p);  // Pointer becomes dangling
4    ...
5    p->func(); // Use-after-free
```

Listing 2.3: Use-after-free vulnerability which can be exploited to hijack the control-flow of the program

2.2.2 Use of Uninitialized Variables

Variables have an *indeterminate value* until they are initialized [78, 79]. C++14 allows this indeterminate value to propagate to other variables if both the source and destination variables have an unsigned narrow character type. Any other use of an uninitialized variable results in undefined behavior.

The effects of this undefined behavior depend on many factors, including the compiler and compiler flags that were used to compile the program. In most cases, indeterminate values are in fact the (partial) contents of previously deallocated variables that occupied the same memory range as the uninitialized variable. As these previously deallocated variables may sometimes hold security-sensitive values, reads of uninitialized memory may be part of an information leakage attack, as illustrated in Listing 2.4.

2.2.3 Pointer Type Errors

C and C++ support several casting operators and language constructs that can lead memory accesses to misinterpret the data stored in their referents, thereby violating type safety. Pointer

```
struct A { int data[2]; };
struct A *p = (struct A *)malloc(sizeof(struct A));
p->data[0] = 0; // Partial initialization
send_to_untrusted_client(p, sizeof(struct A));
```

Listing 2.4: Use of a partially-initialized variable which becomes vulnerable as the uninitialized value crosses a trust boundary

type errors typically result from unsafe casts. C allows all casts between pointer types, as well as casts between integer and pointer types. The C++ `reinterpret_cast` type conversion operator is similarly not subject to any restrictions. The `static_cast` and `dynamic_cast` operators do have restrictions. `static_cast` forbids pointer to integer casts, and casting between pointers to objects that are unrelated by inheritance. However, it does allow casting of a pointer from a base class to a derived class (also called *downcasting*), as well as all casts from and to the `void*` type. *Bad-casting* (often referred to as *type confusion*) happens when a downcast pointer has neither the run-time type of its referent, nor one of the referent's ancestor types.

```
class Base { virtual void func(); };
class Derived : public Base { public: int extra; };
Base b[2];
Derived *d = static_cast<Derived *>(&b[0]); // Bad-casting
d->extra = ...; // Type-unsafe, out-of-bounds access, which
// overwrites the vtable pointer of b[1]
```

Listing 2.5: Bad-casting vulnerability leading to a type- and memory-unsafe memory access

To downcast safely, programmers must use the `dynamic_cast` operator, which performs run-time type checks and returns a null pointer if the check fails. Using `dynamic_cast` is entirely optional, however, and introduces additional run-time overhead.

Type errors can also occur when casting between function pointer types. Again, C++'s `reinterpret_cast` and C impose no restrictions on casts between incompatible function pointer types. If a function is called indirectly through a function pointer of the wrong type,

the target function might misinterpret its arguments, which leads to even more type errors.

Finally, C also allows type punning through `union` types. If the program reads from a union through a different member object than the one that was used to store the data, the underlying memory may be misinterpreted. Furthermore, if the member object used for reading is larger than the member object used to store the data, then the upper bytes read from the union will take unspecified values.

2.2.4 Variadic Function Misuse

C/C++ support *variadic functions*, which accept a variable number of *variadic* function arguments in addition to a fixed number of regular function arguments. The variadic function's source code does not specify the number or types of these variadic arguments. Instead, the fixed arguments and the function semantics encode the expected number and types of variadic arguments. Variadic arguments can be accessed and simultaneously typecast using `va_arg`. It is, in general, impossible to statically verify that `va_arg` accesses a valid argument, or that it casts the argument to a valid type. This lack of static verification can lead to type errors, spatial memory safety violations, and uses of uninitialized values.

```
1    char *fmt2; // User-controlled format string
2    sprintf(fmt2, user_input, ...);
3    // prints attacker-chosen stack contents if fmt2 contains
4    // too many format specifiers
5    // or overwrites memory if fmt2 contains %n
6    printf(fmt2, ...);
```

Listing 2.6: Simplified version of CVE-2012-0809; user-provided input was mistakenly used as part of a larger format string passed to a printf-like function

2.2.5 Other Vulnerabilities

There are other operations that may pose security risks in the absence of type and memory safety. Notable examples include overflow errors which may be exploitable when such values are used in memory allocation or pointer arithmetic operations. If an attacker-controlled integer value is used to calculate a buffer size or an array index, the attacker could overflow that value to allocate a smaller buffer than expected (as illustrated in Listing 2.7), or to bypass existing array index checks, thereby triggering an out-of-bounds access.

C/C++ do not define the result of a signed integer overflow, but stipulate that unsigned integers wrap around when they overflow. However, this wrap-around behavior is often unintended and potentially dangerous.

```c
// newsize can overflow depending on len
int newsize = oldsize + len + 100;
newsize *= 2;
// The new buffer may be smaller than len
buf = xmlRealloc(buf, newsize);
memcpy(buf + oldsize, string, len); // Out-of-bounds access
```

Listing 2.7: Simplified version of CVE-2017-5029; a signed integer overflow vulnerability that can lead to spatial memory safety violation

Undefined behaviors such as signed integer overflows pose additional security risks when compiler optimizations are enabled. In the presence of potential undefined behavior, compilers are allowed to assume that the program will never reach the conditions under which this undefined behavior is triggered. Moreover, the compiler can perform further optimization based on this assumption [94]. Consequently, the compiler does not have to statically verify that the program is free of potential undefined behavior, and the compiler is not obligated to generate code that is capable of recognizing or mitigating undefined behavior.

The problem with this rationale is that optimizations based on the assumption that the program is free from undefined behavior can sometimes lead the compiler to omit security

checks. In CVE-2009-1897, for example, GCC infamously omitted a null pointer check from one of the Linux kernel drivers, which led to a privilege escalation vulnerability [109]. Compiler developers regularly add such aggressive optimizations to their compilers. Some people therefore refer to undefined behavior as *time bombs* [49].

```
1    struct sock *sk = tun->sk; // Compiler assumes tun is not
2    // a null pointer
3    if (!tun) // Check is optimized out
4    return POLLERR;
```

Listing 2.8: Simplified version of CVE-2009-1897; dereferencing a pointer lets the compiler safely assume that the pointer is non-null

2.3 Side-Channel Attacks

Attackers by observing side effects of the computation performed can deduce some of the data that is operated on by the microarchitecture. Hardware side-channel attacks have been studied for a long time and several mitigations have been proposed to prevent against this attack vector such as ensuring algorithms have a constant runtime. However, there has been a renewed interest in software based side-channel attacks in the recent years. In their seminal work Kocher et al. [86] first described how to exploit CPU caches to break cryptographic algorithms. Several variants of attacks that exploit cache side-channels have been published in literature [120, 125, 166, 65, 66, 97] since. In the recent past, software based side-channel attacks have been used for building attacks that can break exploit mitigations such as ASLR [64].

2.3.1 Transient Execution Attacks

Transient-execution attacks are a class of microarchitectural attacks which exploit out-of-order and speculative execution of modern CPUs to leak program data. They rely on microarchitectural operations that were not originally intended to be part of the execution but were carried out by the microarchitectural implementations for performance. Though these unintended operations are never committed to architectural state, they leave side effects which can then be inferred using traditional side-channel attacks. The execution of these unintended operations whose results are eventually discarded is termed transient execution.

Spectre and Spectre Variant 2

Spectre attacks exploit incorrect control-flow and data-flow predictions of CPU. Mispredictions trigger transient execution of instructions that perform unauthorized access to program data as these operations were not intended by the control-flow of the program. By triggering these mispredictions carefully, adversaries can encode the accessed data in the microarchitectural state (e.g. CPU cache). Then a traditional side-channel attack can be employed to extract the encoded data from the microarchitectural state.

Spectre attacks transiently bypass security measures such as bounds checking, memory stores and function calls etc. to perform unauthorized access to the target data. Spectre adversary models involve several variants. In chapter 4, we use the *Spectre Variant 2* to leak randomization secret that is part of a Data Space Randomization protected program and explain specifically this Spectre variant here below.

The idea behind Spectre V2 is similar to that of return-oriented programming and the attacker relies on a target *Spectre Gadget* that is present in the victim process' address space. In this variant, the goal of the attacker is to execute this gadget transiently. Unlike ROP, there is

no need for a software vulnerability to carry out this Spectre attack variant. The attacker instead poisons the Branch Target Buffer (BTB) to trigger a misprediction of a branch from an indirect branch instruction to the address of the target gadget. The transient execution of the gadget will eventually be reverted by the CPU. However, gadgets can be used to encode security sensitive information to the cache which then can be extracted using a traditional cache side-channel attack.

Meltdown and Meltdown-PK

Out-of-order execution of instructions after an exception can also lead to unauthorized access to data. Meltdown attacks are a class of transient execution attacks that rely on exception out-of-order execution scenario. CPUs in-order execution retirement mechanism will eventually revert the out-of-order instructions added to the pipeline. But execution of these instructions just as we saw in Spectre style attacks leave traces in microarchitectural side-channels and can be extracted. Memory Protection Key (MPK) hardware feature on Intel Skylake-SP processors facilitate modification of memory page access permissions from user space (no need for syscall/hypercall). Canella et al. demonstrated that they could successfully read values from memory protected by Intel's Memory Protection Keys (MPK) using a Meltdown variant they term *Meltdown-PK* [23]. Meltodown-PK variant exploits side-effects of out-of-order execution on MPK. Meltdown-PK is a severe form of exploit and there are no software workarounds like the ones for privilege level Meltdown variants.

Chapter 3

A comprehensive approach to fixing variadic vulnerabilities

Programming languages such as C and C++ support variadic functions, i.e., functions that accept a variable number of arguments (e.g., `printf`). In this Chapter, we address the variadic function misuse memory corruption errors.

Variadic functions are flexible, but are inherently type-unsafe. In fact, the semantics and parameters of variadic functions are defined implicitly by their implementation. It is left to the programmer to ensure that the caller and callee follow this implicit specification, without the help of a static type checker. An adversary can take advantage of a mismatch between the argument types used by the caller of a variadic function and the types expected by the callee to violate the language semantics and to tamper with memory. Format string attacks are the most popular example of such a mismatch.

Indirect function calls can be exploited by an adversary to divert execution through illegal paths. CFI restricts call targets according to the function prototype which, for variadic functions, does not include all the actual parameters. However, as shown by our case study,

current CFI implementations are mainly limited to non-variadic functions and fail to address this potential attack vector. Defending against such an attack requires a stateful dynamic check.

We present a compiler based sanitizer to effectively type-check and thus prevent any attack via variadic functions (when called directly or indirectly). The key idea is to record metadata at the call site and verify parameters and their types at the callee whenever they are used at runtime. Our evaluation shows that our approach is (i) practically deployable as the measured overhead is negligible (0.45%) and (ii) effective as we show in several case studies.

3.1 Motivation

We present a new attack against widely deployed mitigations through variadic functions. Variadic functions (such as `printf`) accept a varying number of arguments with varying argument types. To implement variadic functions, the programmer implicitly encodes the argument list in the semantics of the function and has to make sure the caller and callee adhere to this implicit contract. In `printf`, the expected number of arguments and their types are encoded implicitly in the format string, the first argument to the function. Another frequently used scheme iterates through parameters until a condition is reached (e.g., a parameter is `NULL`). Listing 3.1 shows an example of a variadic function. If an adversary can violate the implicit contract between caller and callee, an attack may be possible.

In the general case, it is impossible to enumerate the arguments of a variadic function through static analysis techniques. In fact, their number and types are intrinsic in how the function is defined. This limitation enables (or facilitates) two attack vectors against variadic functions. First, attackers can hijack indirect calls and thereby call variadic functions over control-flow edges that are never taken during any legitimate execution of the program. Variadic functions

that are called in this way may interpret the variadic arguments differently than the function for which these arguments were intended, and thus violate the implicit caller-callee contract. CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type signature of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions.

A second attack vector involves overwriting a variadic function's arguments directly. Such attacks do not violate the intended control flow of a program and thus bypass all of the widely deployed defense mechanisms. Format string attacks are a prime example of such attacks. If an adversary can control the format string passed to, e.g., `printf`, she can control how all of the following parameters are interpreted, and can potentially leak information from the stack, or read/write to arbitrary memory locations.

The attack surface exposed by variadic functions is significant. We analyzed popular software packages, such as Firefox, Chromium, Apache, CPython, nginx, OpenSSL, Wireshark, the SPEC CPU2006 benchmarks, and the FreeBSD base system, and found that variadic functions are ubiquitous. We also found that many of the variadic function calls in these packages are indirect. We therefore conclude that both attack vectors are realistic threats. The underlying problem that enables attacks on variadic functions is the lack of type checking. Variadic functions generally do not (and cannot) verify that the number and type of arguments they expect matches the number and type of arguments passed by the caller. We present our solution, a compiler-based, dynamic sanitizer that tackles this problem by enforcing type checks for variadic functions at run-time. Each argument that is retrieved in a variadic function is type checked, enforcing a strict contract between caller and callee so that (i) a maximum of the passed arguments can be retrieved and (ii) the type of the arguments used at the callee are compatible with the types passed by the caller. Our mechanism can be used in two operation modes: as a runtime monitor to protect programs against attacks and as sanitizer to detect type mismatches during program testing.

We have implemented our prototype, HexVASAN, on top of the LLVM compiler framework, instrumenting the compiled code to record the types of each argument of a variadic function at the call site and to check the types whenever they are retrieved. Our prototype implementation is light-weight, resulting in negligible (0.45%) overhead for SPEC CPU2006. Our approach is general as we show by recompiling the FreeBSD base system and effective as shown through several exploit case studies (e.g., a format string vulnerability in `sudo`).

We present the following contributions:

- Design and implementation of a variadic function sanitizer on top of LLVM;
- A case study on large programs to show the prevalence of direct and indirect calls to variadic functions;
- Several exploit case studies and CFI bypasses using variadic functions.

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int add(int start, ...) {
5      int next, total = start;
6      va_list list;
7      va_start(list, start);
8      do {
9          next = va_arg(list, int);
10         total += next;
11     } while (next != 0);
12     va_end(list);
13     return total;
14 }
15
16 int main(int argc, const char *argv[]) {
17     printf("%d\n", add(5, 1, 2, 0));
18     return 0;
19 }
```

Listing 3.1: Example of a variadic function in C. The function `add` takes a non-variadic argument `start` (to initialize an accumulator variable) and a series of variadic `int` arguments that are added until the terminator value `0` is met. The final value is returned.

3.2 Background

Variadic functions are used ubiquitously in C/C++ programs. Here we introduce details about their use and implementation on current systems, the attack surface they provide, and how adversaries can abuse them.

3.2.1 Variadic functions

Variadic functions (such as the `printf` function in the C standard library) are used in C to maximize the flexibility in the interface of a function, allowing it to accept a number of arguments unknown at compile-time. These functions accept a variable number of arguments, which do not necessarily have fixed types. An example of a variadic function is shown in Listing 3.1. The function `add` accepts one mandatory argument (`start`) and a varying number of additional arguments, which are marked by the ellipsis (`...`) in the function definition.

The C standard defines several macros that portable programs may use to access variadic arguments [96]. `stdarg.h`, the header that declares these macros, defines an opaque type, `va_list`, which stores all information required to retrieve and iterate through variadic arguments. In our example, the variable `list` of type `va_list` is initialized using the `va_start` macro. The `va_arg` macro retrieves the next variadic argument from the `va_list`, updating `va_list` to point to the next argument as a side effect. Note that, although the programmer must specify the expected type of the variadic argument in the call to `va_arg`, the C standard does not require the compiler to verify that the retrieved variable is indeed of that type. `va_list` variables must be released using a call to the `va_end` macro so that all of the resources assigned to the list are deallocated.

`printf` is an example of a more complex variadic function which takes a format string as its first argument. This format string implicitly encodes information about the number

of arguments and their type. Implementations of `printf` scan through this format string several times to identify all format arguments and to recover the necessary space in the output string for the specified types and formats. Interestingly, arguments do not have to be encoded sequentially but format strings allow out-of-order access to arbitrary arguments. This flexibility is often abused in format string attacks to access arbitrary stack locations.

3.2.2 Variadic functions ABI

The C standard does not define the calling convention for variadic functions, nor the exact representation of the `va_list` structure. This information is instead part of the ABI of the target platform.

x86-64 ABI. The AMD64 System V ABI [103], which is implemented by x86-64 GNU/Linux platforms, dictates that the caller of a variadic function must adhere to the normal calling conventions when passing arguments. Specifically, the first six non-floating point arguments and the first eight floating point arguments are passed through CPU registers. The remaining arguments, if any, are passed on the stack. If a variadic function accepts five mandatory arguments and a variable number of variadic arguments, then all but one of these variadic arguments will be passed on the stack. The variadic function itself moves the arguments into a `va_list` variable using the `va_start` macro. The `va_list` type is defined as follows:

```
typedef struct {
  unsigned int gp_offset;
  unsigned int fp_offset;
  void *overflow_arg_area;
  void *reg_save_area;
} va_list[1];
```

`va_start` allocates on the stack a `reg_save_area` to store copies of all variadic arguments that were passed in registers. `va_start` initializes the `overflow_arg_area` field to point to the first variadic argument that was passed on the stack. The `gp_offset` and `fp_offset` fields are the

offsets into the `reg_save_area`. They represent the first unused variadic argument that was passed in a general purpose register or floating point register respectively.

The `va_arg` macro retrieves the first unused variadic argument from either the `reg_save_area` or the `overflow_arg_area`, and either it increases the `gp_offset`/`fp_offset` field or moves the `overflow_arg_area` pointer forward, to point to the next variadic argument.

Other architectures. Other architectures may implement variadic functions differently. On 32-bit x86, for example, all variadic arguments must be passed on the stack (pushed right to left), following the `cdecl` calling convention used on GNU/Linux. The variadic function itself retrieves the first unused variadic argument directly from the stack. This simplifies the implementation of the `va_start`, `va_arg`, and `va_end` macros, but it generally makes it easier for adversaries to overwrite the variadic arguments.

3.2.3 Variadic attack surface

When calling a variadic function, the compiler statically type checks all non-variadic arguments but does not enforce any restriction on the type or number of variadic arguments. The programmer must follow the implicit contract between caller and callee that is only present in the code but never enforced explicitly. Due to this high flexibility, the compiler cannot check arguments statically. This lack of safety can lead to bugs where an adversary achieves control over the callee by modifying the arguments, thereby influencing the interpretation of the passed variadic arguments.

Modifying the argument or arguments that control the interpretation of variadic arguments allows an adversary to change the behavior of the variadic function, causing the callee to access additional or fewer arguments than specified and to change the interpretation of their types.

An adversary can influence variadic functions in several ways. First, if the programmer forgot to validate the input, the adversary may directly control the arguments to the variadic function that controls the interpretation of arguments. Second, the adversary may use an arbitrary memory corruption elsewhere in the program to influence the argument of a variadic function.

Variadic functions can be called statically or dynamically. Direct calls would, in theory, allow some static checking. Indirect calls (e.g., through a function pointer), where the target of the variadic function is not known, do not allow any static checking. Therefore, variadic functions can only be protected through some form of runtime checker that considers the constraints of the call site and enforces them at the callee side.

3.2.4 Format string exploits

Format string exploits are a perfect example of corrupted variadic functions. An adversary that gains control over the format string used in `printf` can abuse the `printf` function to leak arbitrary data on the stack or even resort to arbitrary memory corruption (if the pointer to the target location is on the stack). For example, a format string vulnerability in the smbclient utility (CVE-2009-1886) [113] allows an attacker to gain control over the Samba file system by treating a filename as format string. Also, in PHP 7.x before 7.0.1, an error handling function in zend_execute_API.c allows an attacker to execute arbitrary code by using format string specifiers as class name (CVE-2015-8617) [1].

Information leaks are simple: an adversary changes the format string to print the desired information that resides somewhere higher up on the stack by employing the desired format string specifiers. For arbitrary memory modification, an adversary must have the target address encoded somewhere on the stack and then reference the target through the `%n` modifier, writing the number of already written bytes to that memory location.

The GNU C standard library (*glibc*) enforces some protection against format string attacks by checking if a format string is in a writable memory area [82]. For format strings, the *glibc* `printf` implementation opens `/proc/self/maps` and scans for the memory area of the format string to verify correct permissions. Moreover, a check is performed to ensure that all arguments are consumed, so that no out-of-context stack slots can be used in the format string exploit. These defenses stop some attacks but do not mitigate the underlying problem that an adversary can gain control over the format string. Note that this heavyweight check is only used if the format string argument *may* point to a writable memory area at compile time. An attacker may use memory corruption to redirect the format string pointer to an attacker-controlled area and fall back to a regular format string exploit.

3.3 Threat model

Programs frequently use variadic functions, either in the program itself or as part of a shared library (e.g., `printf` in the C standard library). We assume that the program contains an arbitrary memory corruption, allowing the adversary to modify the arguments to a variadic function and/or the target of an indirect function call, targeting a variadic function.

Our target system deploys existing defense mechanisms like DEP, ASLR, and a strong implementation of CFI, protecting the program against code injection and control-flow hijacking. We assume that the adversary cannot modify the metadata of our runtime monitor. Protecting metadata is an orthogonal engineering problem and can be solved through, e.g., masking (`and`-ing every memory access), segmentation (for x86-32), protecting the memory region [26], or randomizing the location of sensitive data. Our threat model is a realistic scenario for current attacks and defenses.

3.4 Design

Our solution monitors calls to variadic functions and checks for type violations. Since the semantics of how arguments should be interpreted by the function are intrinsic in the logic of the function itself, it is, in general, impossible to determine the number and type of arguments a certain variadic function accepts. For this reason, we instrument the code generated by extending the compiler so that a check is performed at runtime. This check ensures that the arguments consumed by the variadic function match those passed by the caller.

The high level idea is the following: record metadata about the supplied argument types at the call site and verify that the extracted arguments match in the callee. The number of arguments and their types is always known at the call site and can be encoded efficiently. In the callee this information can then be used to verify individual arguments when they are accessed. To implement such a sanitizer, we must design a metadata store, a pass that instruments call sites, a pass that instruments callers, and a runtime library that manages the metadata store and performs the run-time type verification. Our runtime library gracefully terminates the program whenever a mismatch is detected and generates detailed information about the call site and the mismatched arguments.

3.4.1 Analysis and Instrumentation

We designed our prototype, HexVASAN, as a compiler pass to be run in the compilation pipeline right after the C/C++ frontend. The instrumentation collects a set of statically available information about the call sites, encodes it in the LLVM module, and injects calls to our runtime to perform checks during program execution.

Figure 3.1 provides an overview of the compilation pipeline when our solution is enabled. Source files are first parsed by the C/C++ frontend which generates the intermediate represen-

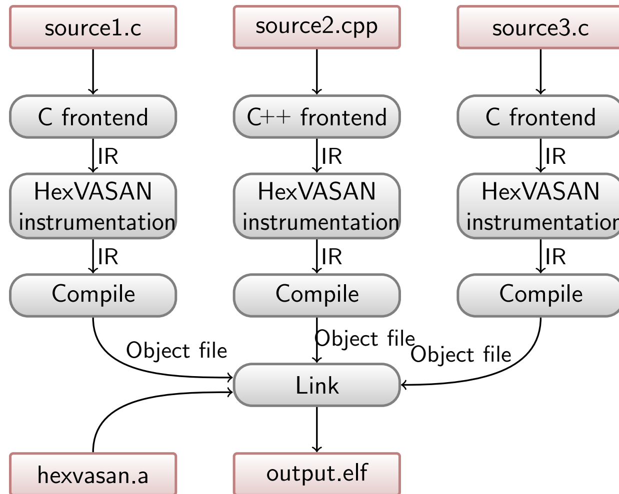

Figure 3.1: Overview of our compilation pipeline. Our instrumentation passes run right after the C/C++ frontend, while our runtime library, `hexvasan.a`, is merged into the final executable at link time.

tation on which our instrumentation runs. The normal compilation then proceeds, generating instrumented object files. These object files, along with the HexVASAN runtime library, are then passed to the linker, which creates the instrumented program binary.

3.4.2 Runtime support

Our instrumentation augments every `va_list` in the original program with the type information generated by our instrumentation pass, and uses this type information to perform run-time type checking on any variadic argument accessed through `va_arg`. By managing the type information in a metadata store, and by maintaining a mapping between `va_lists` and their associated type information, our approach remains fully compatible with the platform ABI. This design also supports interfacing between instrumented programs and non-instrumented libraries.

Our runtime support library manages the type information in two data structures. The core data structure, called the *variadic list map* (VLM), associates `va_list` structures with the type information produced by our instrumentation, and with a counter to track the index of

the last argument that was read from the list. A second data structure, the *variadic call stack* (VCS), allows callers of variadic functions to store type information of variadic arguments until the callee initializes the `va_list`.

Each variadic call site is instrumented with a call to `pre_call`, that prepares the information about the call site (a *variadic call site descriptor* or VCSD), and a call to `post_call`, that cleans it up. For each variadic function, the `va_start` calls are instrumented with `list_init`, while `va_copy`, whose purpose is to clone a `va_list`, is instrumented through `list_copy`. The two run-time functions will allocate the necessary data structures to validate individual arguments. Calls to `va_end` are instrumented through `list_end` to free up the corresponding data structures.

Algorithm 1 summarizes the two phases of our analysis and instrumentation pass. The first phase identifies all the calls to variadic functions (both direct and indirect). Note that identifying indirect calls to variadic functions is straight-forward in a compiler framework since, even if the target function is not statically known, its type is. Then, all the parameters passed by that specific call site are inspected and recorded, along with their type in a dedicated VCSD which is stored in read-only global data. At this point, a call to `pre_call` is injected before the variadic function call (with the newly created VCSD as a parameter) and, symmetrically, a call to `post_call` is inserted after the call site.

The second phase identifies all calls to `va_start` and `va_copy`, and consequently, the `va_list` variables in the program. Uses of each `va_list` variable are inspected in an architecture-specific way. Once all uses are identified, we inject a call to `check_arg` before dereferencing the argument (which always resides in memory).

3.4.3 Challenges and Discussion

When designing a variadic function call sanitizer, several issues have to be considered. We highlight details about the key challenges we encountered.

Multiple `va_lists`. Functions are allowed to create multiple `va_lists` to access the same variadic arguments, either through `va_start` or `va_copy` operations. We handle this by storing a VLM entry for each individual `va_list`.

```
input: a module m
/* Phase 1                                                    */
foreach function f in module m do
    foreach variadic call c with n arguments in f do
        vcsd.count ← n;
        foreach argument a of type t do
            vcsd.args.push(t);
        end
        emit call to pre_call(vcsd) before c;
        emit call to post_call() after c;
    end
end
/* Phase 2                                                    */
foreach function f in module m do
    foreach call c to va_start(list) do
        emit call to list_init(&list) after c;
    end
    foreach call c to va_copy(dst, src) do
        emit call to list_copy(&dst, &src) after c;
    end
    foreach call c to va_end(list) do
        emit call to list_free(&list) after c;
    end
    foreach call c to va_arg(list, type) do
        emit call to check_arg(&list, type) before c;
    end
end
```

Algorithm 1: The instrumentation process.

Passing `va_lists` as function arguments. While uncommon, variadic functions are allowed to pass the `va_list` instances they create as arguments to non-variadic functions. This allows non-variadic functions to access variadic arguments of functions higher in the call stack. Our design takes this into account by maintaining a list map (VLM) and by instrumenting all `va_arg` operations, regardless of whether or not they are in a variadic function.

Multi-threading support. We support multiple threads by storing our per-thread runtime state in a thread-local variable.

Metadata format. We use a constant data structure per variadic call site, the VCSD, to hold the number of arguments and a pointer to an array of integers identifying their type. The `check_arg` function therefore only performs two memory accesses, the first to load the number of arguments and the second for the type of the argument currently being checked.

To uniquely identify the data types with an integer, we built a hashing function (described in Algorithm 2) using a set of fixed identifiers for primitive data types and hashing them in different ways depending on how they are aggregated (pointers, `union`, or `struct`). The last hash acts as a terminator marker for aggregate types, which allows us to, e.g., distinguish between {`struct{ int }, int`} and {`struct {struct{ int, int }}`}. Note that an (unlikely) hash collision only results in two different types being accepted as equal. Such a hashing mechanism has the advantage of being deterministic across compilation units, removing the need for keeping a global map of type-unique id pairs. Due to the information loss during the translation from C/C++ to LLVM IR, our type system does not distinguish between signed and unsigned types. The required metadata is static and immutable and we mark it as read-only, protecting it from modification. However, the VCS still needs to be protected through other mechanisms.

Handling floating point arguments. In x86-64 ABI, floating point and non-floating

```
input  : a type t and an initial hash value h
output : the final hash value h
h = hash(h, typeID(t));
switch typeID(t) do
    case AggregateType do
        /* union, struct and pointer */
        foreach c in componentTypes(t) do
            h = hashType(c, h);
        end
    case FunctionType do
        h = hashType(returnType(t), h);
        foreach a in argTypes(t) do
            h = hashType(a, h);
        end
    end
end
h = hash(h, typeID(t));
return h
```

Algorithm 2: Algorithm describing the type hashing function *hashType*. *typeID* returns an unique identifier for each basic type (e.g., 32-bit integer, `double`), type of aggregate type (e.g., `struct`, `union`...) and functions. *hash* is a simple hashing function combining two integers. *componentTypes* returns the components of an aggregate type, *returnType* the return type of a function prototype and *argTypes* the type of its arguments.

point arguments are handled differently. In case of floating point arguments, the first eight arguments are passed in the floating point registers whereas in case of non-floating point the first six are passed in general-purpose registers. We handle both argument types.

Support for aggregate data types. According to AMD64 System V ABI, the caller unpacks the fields of the aggregate data types (structs and unions) if the arguments fit into registers. This makes it hard to distinguish between composite types and regular types – if unpacked they are indistinguishable on the callee side from arguments of these types. We support aggregate data types even if the caller unpacks them.

Attacks preserving number and type of arguments. Our mechanism prevents attacks that change the number of arguments or the types of individual arguments. Format string attacks that only change one modifier can therefore be detected through the type mismatch

even if the total number of arguments remains unchanged.

Non-variadic calls to variadic functions. Consider the following code snippet:

```
typedef void (*non_variadic)(int, int);

void variadic(int, ...) { /* ... */ }

int main() {
  non_variadic function_ptr = variadic;
  function_ptr(1, 2);
}
```

In this case, the function call in `main` to `function_ptr` appears to the compiler as a non-variadic function call, since the type of the function pointer is not variadic. Therefore, our pass will not instrument the call site, leading to potential errors.

To handle such (rare) situations appropriately, we would have to instrument all non-variadic call sites too, leading to an unjustified overhead. Moreover, the code above represents *undefined behavior* in C [77, 6.3.2.3p8] and C++ [76, 5.2.10p6], and might not work on certain architectures where the calling convention for variadic and non-variadic function calls are not compatible. The GNU C compiler emits a warning when a function pointer is cast to a different type, therefore we require the developer to correct the code before applying our solution.

Central management of the global state. To allow our runtime support functions to be linked into the base system libraries, such as the C standard library, we made it a static library. Turning the runtime into a shared library is possible, but would prohibit its use during the early process initialization – until the dynamic linker has processed all of the necessary relocations. Our runtime therefore either needs to be added solely to the C

standard library (so that it is initialized early in the startup process) or the runtime library must carefully use weak symbols to ensure that each symbol is only defined once if multiple libraries are compiled with our countermeasure.

C++ exceptions and `longjmp`. If an exception is raised while executing a variadic function (or one of its callees), the variadic function may not get a chance to clean up the metadata for any `va_lists` it has initialized, nor may the caller of this variadic function get the chance to clean up the type information it has pushed onto the VCS. Other functions manipulating the thread's stack directly, such as `longjmp`, present similar issues.

C++ exceptions can be handled by modifying the LLVM C++ frontend (i.e., `clang`) to inject an object with a lifetime spanning from immediately before a variadic function call to immediately after. Such an object would call `pre_call` in its constructor and `post_call` in the destructor, leveraging the exception handling mechanism to make our implementation exception-safe. Functions like `longjmp` can be instrumented to purge the portions of metadata data structures that correspond to the discarded stack area. We did not observe any such calls in practice and leave the implementation of handling exceptions and `longjump` across variadic functions as future engineering work.

3.5 Implementation

We implemented our prototype, HexVASAN, as a sanitizer for the LLVM compiler framework [91], version 3.9.1. We have chosen LLVM for its robust features on analyzing and transforming arbitrary programs as well as extracting reliable type information. The sanitizer can be enabled from the C/C++ frontend (`clang`) by providing the `-fsanitize=vasan` parameter at compile-time. No annotations or other source code changes are required for HexVASAN. Our sanitizer does not require visibility of whole source code (see Section 3.4.3),

but works on individual compilation units. Therefore link-time optimization (LTO) is not required and thus fits readily into existing build systems. In addition, HexVASAN also supports signal handlers.

HexVASAN consists of two components: a static instrumentation pass and a runtime library. The static instrumentation pass works on LLVM IR, adding the necessary instrumentation code to all variadic functions and their callees. The support library is statically linked to the program and, at run-time, checks the number and type of variadic arguments as they are used by the program. In the following we describe the two components in detail.

Static instrumentation. The implementation of the static instrumentation pass follows the description in Section 3.4. We first iterate through all functions, looking for `CallInst` instructions targeting a variadic function (either directly or indirectly), then we inspect them and create for each one of them a read-only `GlobalVariable` of type `vcsd_t`. As shown in Listing 3.2, `vcsd_t` is composed by an unsigned integer representing the number of arguments of the considered call site and a pointer to an array (another `GlobalVariable`) with an integer element for each argument of `type_t`. `type_t` is an integer uniquely identifying a data type obtained using the *hashType* function presented in Algorithm 2. At this point a call to `pre_call` is injected before the call site, with the newly create VCSD as a parameter, and a call to `post_call` is injected after the call site.

During the second phase, we first identify all `va_start`, `va_copy`, and `va_end` operations in the program. In the IR code, these operations appear as calls to the LLVM intrinsics `llvm.va_start`, `llvm.va_copy`, and `va_end`. We instrument the operations with calls to our runtime's `list_init`, `list_copy`, and `list_free` functions respectively. We then proceed to identify `va_arg` operations. Although the LLVM IR has a dedicated `va_arg` instruction, it is not used on any of the platforms we tested. The `va_list` is instead accessed directly. Our identification of `va_arg` is therefore platform-specific. On x86-64, our primary target, we identify `va_arg` by recognizing accesses to the `gp_offset` and `fp_offset` fields in the x86-64

```
struct vcsd_t {
  unsigned count;
  type_t *args;
};
thread_local stack<vcsd_t *> vcs;
thread_local map<va_list *, pair<vcsd_t *, unsigned>> vlm;
void pre_call(vcsd_t *arguments) {
  vcs.push_back(arguments);
}
void post_call() {
  vcs.pop_back();
}
void list_init(va_list *list_ptr) {
  vlm[list_ptr] = { vcs.top(), 0 };
}
void list_free(va_list *list_ptr) {
  vlm.erase(list_ptr);
}
void check_arg(va_list *list_ptr, type_t type) {
  pair<vcsd_t *, unsigned> &args = vlm[list_ptr];
  unsigned index = args.second++;
  assert(index < args.first->count);
  assert(args.first->args[index] == type);
}
int add(int start, ...) {
  /* ... */
  va_start(list, start);
  list_init(&list);
  do {
    check_arg(&list, typeid(int));
    total += va_arg(list, int);
  } while (next != 0);
  va_end(list);
  list_free(&list);
  /* ... */
}
const vcsd_t main_add_vcsd = {
  .count = 3,
  .args = {typeid(int), typeid(int), typeid(int)}
};
int main(int argc, const char *argv[]) {
  /* ... */
  pre_call(&main_add_vcsd);
  int result = add(5, 1, 2, 0);
  post_call();
  printf("%d\n", result);
  /* ... */
}
```

Listing 3.2: Simplified C++ representation of the instrumented code for Listing 3.1.

version of the va_list structure (see Section 3.2.2). The fp_offset field is accessed whenever the program attempts to retrieve a floating point argument from the list. The gp_offset field is accessed to retrieve any other types of variadic arguments. We insert a call to our runtime's check_arg function before the instruction that accesses this field.

Listing 3.2 shows (in simplified C) how the code in Listing 3.1 would be instrumented by our sanitizer.

Dynamic variadic type checking. The entire runtime is implemented in plain C code, as this allows it to be linked into the standard C library without introducing a dependency to the standard C++ library. The VCS is implemented as a thread-local stack, and the VLM as a thread-local hash map. The pre_call and post_call functions push and pop type information onto and from the VCS. The list_init function inserts a new entry into the VLM, using the top element on the stack as the entry's type information and initializing the counter for consumed arguments to 0.

check_arg looks up the type information for the va_list being accessed in the VLM and checks if the requested argument exists (based on the counter of consumed arguments), and if its type matches the one provided by the caller. If either of these checks fails, execution is aborted, and the runtime will generate an error message such as the one shown in Listing 3.3. As a consequence, the pointer to the argument is never read or written, since the pointer to it is never dereferenced.

3.6 Evaluation

In this section we present a case study on variadic function based attacks against state-of-the-art CFI implementations. Next, we evaluate the effectiveness of our approach as an exploit mitigation technique. Then, we evaluate the overhead introduced by our prototype

```
Error: Type Mismatch
Index is 1
Callee Type : 43 (32-bit Integer)
Caller Type : 15 (Pointer)
Backtrace:
[0] 0x4019ff <__vasan_backtrace+0x1f> at test
[1] 0x401837 <__vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
```

Listing 3.3: Error message reported by HexVASAN

implementation on the SPEC CPU2006 integer (CINT2006) benchmarks, on Firefox using standard JavaScript benchmarks, and on micro-benchmarks. We also evaluate how widespread the usage of variadic functions is in SPEC CPU2006 and in Firefox 51.0.1, Chromium 58.0.3007.0, Apache 2.4.23, CPython 3.7.0, nginx 1.11.5, OpenSSL 1.1.1, Wireshark 2.2.1, and the FreeBSD 11.0 base system.

Note that, along with testing the aforementioned software, we also developed an internal set of regression tests. Our regression tests allow us to verify that our sanitizer correctly catches problematic variadic function calls, and does not raise false alarms for benign calls. The test suite explores corner cases, including trying to access arguments that have not been passed and trying to access them using a type different from the one used at the call site.

3.6.1 Case study: CFI effectiveness

One of the attack scenarios we envision is that an attacker controls the target of an indirect call site. If the intended target of the call site was a variadic function, the attacker could illegally call a different variadic function that expects different variadic arguments than the intended target (yet shares the types for all non-variadic arguments). If the intended target

Intended Target	Actual target: Prototype	Actual target: Address Taken?	LLVM CFI	pi CFI	CCFI	VTV	CFG	Hex VASAN
Variadic	Same	Yes	✗	✗	✗	✗	✗	✓
		No	✗	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
Non-variadic	Same	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✓	✗	✗	✓
Original	Overwritten Arguments		✗	✗	✗	✗	✗	✓

Table 3.1: Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection.

of the call site was a non-variadic function, the attacker could call a variadic function that interprets some of the intended target's arguments as variadic arguments.

All existing CFI mechanisms allow such attacks to some extent. The most precise CFI mechanisms, which rely on function prototypes to classify target sets (e.g., LLVM-CFI, piCFI, or VTV) will allow all targets with the same prototype, possibly restricting to the subset of functions whose addresses are taken in the program. This is problematic for variadic functions, as only non-variadic types are known statically. For example, if a function of type `int (*)(int, ...)` is expected to be called from an indirect call site, then precise CFI schemes allow calls to all other variadic functions of that type, even if those other functions expect different types for the variadic arguments.

A second way to attack variadic functions is to overwrite their arguments directly. This happens, for example, in format string attacks, where an attacker can overwrite the format string to cause misinterpretation of the variadic arguments. Our implementation prototype

detects both of these attacks when the callee attempts to retrieve the variadic arguments using the `va_arg` macro described in Section 3.2.1. Checking and enforcing the correct types for variadic functions is only possible at runtime and any sanitizer must resort to run-time checks to do so. CFI mechanisms must therefore be extended with a dynamic variadic argument type check mechanism that resembles our approach to detect such violations. To show that our tool can complement CFI, we create test programs containing several variadic functions and one non-variadic function. The definitions of these functions are shown below.

```
1 int sum_ints(int n, ...);
2 int avg_longs(int n, ...);
3 int avg_doubles(int n, ...);
4 void print_longs(int n, ...);
5 void print_doubles(int n, ...);
6 int square(int n);
```

This program contains one indirect call site from which only the `sum_ints` function can be called legally, and one indirect call site from which only the `square` function can be legally called. We also introduce a memory corruption vulnerability which allows us to override the target of both indirect calls.

We constructed the program such that `sum_ints`, `avg_longs`, `print_longs`, and `square` are all address-taken functions. The `avg_doubles` and `print_doubles` functions are not address-taken.

Functions `avg_longs`, `avg_doubles`, `print_longs`, and `print_doubles` all expect different variadic argument types than function `sum_ints`. Functions `sum_ints`, `avg_longs`, `avg_doubles`, and `square` do, however, all have the same non-variadic prototype (`int (*)(int)`).

We compiled six versions of the test program, instrumenting them with, respectively, HexVASAN, LLVM 3.9 Forward-Edge CFI [153], Per-Input CFI [117], CCFI [102], GCC 6.2's VTV [153] and Visual C++ Control Flow Guard [106]. In each version, we first built an

attack involving a variadic function, by overriding the indirect call sites with a call to each of the variadic functions described above. We then also tested overwriting the arguments of the `sum_ints` function, without overwriting the indirect call target. Table 3.1 shows the detection results.

LLVM Forward-Edge CFI allows calls to `avg_longs` and `avg_doubles` from the `sum_ints` indirect call site because these functions have the same static type signature as the intended call target. This implementation of CFI does not allow calls to variadic functions from non-variadic call sites, however.

CCFI only detects calls to `print_doubles`, a function that is not address-taken and has a different non-variadic prototype than `square`, from the `square` call site. It allows all of the other illegal calls.

GCC VTV, and Visual C++ CFG allow all of the illegal calls, even if the non-variadic type signature does not match that of the intended call target.

pi-CFI allows calls to the `avg_longs` function from the `sum_ints` indirect call site. `avg_longs` is address-taken and it has the same static type signature as the intended call target. pi-CFI does not allow illegal calls to non-address-taken functions or functions with different static type signatures. pi-CFI also does not allow calls to variadic functions from non-variadic call sites.

All implementations of CFI allow direct overwrites of variadic arguments, as long as the original control flow of the program is not violated.

3.6.2 Exploit Detection

To evaluate the effectiveness of our tool as a real-world exploit detector, we built a version of `sudo` 1.8.3 with our variadic argument type checking instrumentation. `sudo` allows authorized users to execute shell commands as another user, often one with a higher privilege level on the system. If compromised, `sudo` can escalate the privileges of non-authorized users, making it a popular target for attackers. Versions 1.8.0 through 1.8.3p1 of `sudo` contained a format string vulnerability (CVE-2012-0809) that allowed exactly such a compromise. This vulnerability could be exploited by passing a format string as the first argument (`argv[0]`) of the `sudo` program. One such exploit was shown to bypass ASLR, DEP, and glibc's FORTIFY_SOURCE protection [53]. In addition, we were able to verify that GCC 5.4.0 and clang 3.8.0 fail to catch this exploit, even when annotating the vulnerable function with the `format` function attribute [5] and setting the compiler's format string checking (`-Wformat`) to the highest level.

Although it is `sudo` itself that calls the format string function (`fprintf`), our instrumentation can only detect the violation on the callee side. We therefore had to build hardened versions of not just the `sudo` binary itself, but also the C library. We chose to do this on the FreeBSD platform, as its standard C library can be easily built using LLVM, and our implementation therefore readily fits into the FreeBSD build process. As expected, our solution does detect any exploit that triggers the vulnerability, producing the error message shown in Listing 3.4.

3.6.3 Prevalence of variadic functions

To collect variadic function usage in real software, we extended our instrumentation mechanism to collect statistics about variadic functions and their calls. As shown in Table 3.2 and Table 3.3, for each program, we collect:

Call sites. The number of function calls targeting variadic functions. We report the total

```
$ ln -s /usr/bin/sudo %x%x%x%x
$ ./%x%x%x%x -D9 -A
--------------------------
Error: Index greater than Argument Count
Index is 1
Backtrace:
[0] 0x4053bf <__vasan_backtrace+0x1f> at sudo
[1] 0x405094 <__vasan_check_index+0xf4> at sudo
[2] 0x8015dce24 <__vfprintf+0x2174> at libc.so
[3] 0x8015dac52 <vfprintf_l+0x212> at libc.so
[4] 0x8015daab3 <vfprintf_l+0x73> at libc.so
[5] 0x40bdaf <sudo_debug+0xdf> at sudo
[6] 0x40ada3 <main+0x6c3> at sudo
[7] 0x40494f <_start+0x17f> at sudo
```

Listing 3.4: Exploit detection in sudo.

	Call sites			Func.			Ratio	
Program	Tot.	Ind.	%	Tot.	A.T.	Proto	Tot.	A.T.
Firefox	30225	1664	5.5	421	18	241	1.75	0.07
Chromium	83792	1728	2.1	794	44	396	2.01	0.11
FreeBSD	189908	7508	3.9	1368	197	367	3.73	0.53
Apache	7121	0	0	94	29	41	2.29	0.71
CPython	4183	0	0	382	0	38	10.05	0.00
nginx	1085	0	0	26	0	14	1.86	0.00
OpenSSL	4072	1	0.02	23	0	15	1.53	0.00
Wireshark	37717	0	0	469	1	110	4.26	0.01

Table 3.2: Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect); % shows the percentage of variadic call sites. The fifth and sixth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions.

number and how many of them are indirect, since they are of particular interest for an attack scenario where the adversary can override a function pointer.

Variadic functions. The number of variadic functions. We report their total number and

Program	Call sites			Func.		Proto	Ratio	
	Tot.	Ind.	%	Tot.	A.T.		Tot.	A.T.
perlbench	1460	1	0.07	60	2	18	3.33	0.11
bzip2	85	0	0	3	0	3	1.00	0.00
gcc	3615	55	1.5	125	0	31	4.03	0.00
mcf	29	0	0	3	0	3	1.00	0.00
milc	424	0	0	21	0	8	2.63	0.00
namd	485	0	0	24	2	8	3.00	0.25
gobmk	2911	0	0	35	0	8	4.38	0.00
soplex	6	0	0	2	1	2	1.00	0.50
povray	1042	40	3.8	45	10	16	2.81	0.63
hmmer	671	7	1	9	1	5	1.80	0.20
sjeng	253	0	0	4	0	3	1.33	0.00
libquantum	74	0	0	91	0	7	13.00	0.00
h264ref	432	0	0	85	5	13	6.54	0.38
lbm	11	0	0	3	0	2	1.50	0.00
omnetpp	340	0	0	48	23	19	2.53	1.21
astar	42	0	0	4	1	4	1.00	0.25
sphinx3	731	0	0	20	0	5	4.00	0.00
xalancbmk	19	0	0	4	2	4	1.00	0.50

Table 3.3: Statistics of Variadic Functions for SPEC 2006 Benchmarks.

how many of them have their address taken, since CFI mechanism cannot prevent functions with their address taken from being reachable from indirect call sites.

Variadic prototypes. The number of distinct variadic function prototypes in the program.

Functions-per-prototype. The average number of variadic functions sharing the same prototype. This measures how many targets are available, on average, for each indirect call sites targeting a specific prototype. In practice, this the average number of permitted destinations for an indirect call site in the case of a perfect CFI implementation. We report

this value both considering all the variadic functions and only those whose address is taken.

Interestingly, each benchmark we analyzed contains calls to variadic functions and several programs (Firefox, OpenSSL, perlbench, gcc, povray, and hmmer) even contain indirect calls to variadic functions. In addition to *calling* variadic functions, each benchmark also *defines* numerous variadic functions (421 for Firefox, 794 for Chromium, 1368 for FreeBSD, 469 for Wireshark, and 382 for CPython). Variadic functions are therefore prevalent and used ubiquitously in software. Adversaries have plenty of opportunities to modify these calls and to attack the implicit contract between caller and callee. The compiler is unable to enforce any static safety guarantees when calling these functions, either for the number of arguments, nor their types. In addition, many of the benchmarks have variadic functions that are called indirectly, often with their address being taken. Looking at Firefox, a large piece of software, the numbers are even more staggering with several thousand indirect call sites that target variadic functions and 241 different variadic prototypes.

The prevalence of variadic functions leaves both a large attack surface for attackers to either redirect variadic calls to alternate locations (even if defense mechanisms like CFI are present) or to modify the arguments so that callees misinterpret the supplied arguments (similar to extended format string attacks).

In addition, the compiler has no insight into these functions and cannot statically check if the programmer supplied the correct parameters. Our sanitizer identified three interesting cases in `omnetpp`, one of the SPEC CPU2006 benchmarks that implements a discrete event simulator. The benchmark calls a variadic functions with a mismatched type, where it expects a `char *` but receives a NULL, which has type `void *`. Listing 3.5 shows the offending code.

We also identify a bug in SPEC CPU2006's `perlbench`. This benchmark passes the result of a subtraction of two character pointers as an argument to a variadic function. At the call site, this argument is a machine word-sized integer (i.e., 64-bits integer on our test platform).

```
static sEnumBuilder _EtherMessageKind( "EtherMessageKind",
    JAM_SIGNAL, "JAM_SIGNAL",
    ETH_FRAME, "ETH_FRAME",
    ETH_PAUSE, "ETH_PAUSE",
    ETHCTRL_DATA, "ETHCTRL_DATA",
    ETHCTRL_REGISTER_DSAP, "ETHCTRL_REGISTER_DSAP",
    ETHCTRL_DEREGISTER_DSAP, "ETHCTRL_DEREGISTER_DSAP",
    ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
    0, NULL
);
```

Listing 3.5: Variadic violation in omnetpp.

The callee truncates this argument to a 32-bit integer by calling `va_arg(list, int)`. Our implementation prototype reports this (likely unintended) truncation as a violation.

3.6.4 Firefox

We evaluate the performance of our implementation prototype by instrumenting Firefox (51.0.1) and using three different browser benchmark suites: Octane, JetStream, and Kraken. Table 3.4 shows the comparison between the HexVASAN instrumented Firefox and native Firefox. To reduce variance between individual runs, we averaged fifteen runs for each benchmark (after one warmup run). For each run we started Firefox, ran the benchmark,

Benchmark		Native	HexVASAN
Octane	AVERAGE	31241.80	30907.73
	STDDEV	2449.82	2442.82
	OVERHEAD		-1.08%
JetStream	AVERAGE	200.76	198.75
	STDDEV	0.66	1.68
	OVERHEAD		-1.01%
Kraken	AVERAGE [ms]	832.48	832.41
	STDDEV [ms]	7.41	12.71
	OVERHEAD		0.01%

Table 3.4: Performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better.

and closed the browser. HexVASAN incurs only 1.08% and 1.01% overhead for Octane and JetStream respectively and speeds up around 0.01% for Kraken. These numbers are indistinguishable from measurement noise. Octane [4] and JetStream measure the time a test takes to complete and then assign a score that is inversely proportional to the runtime, whereas Kraken [3] measures the speed of test cases gathered from different real-world applications and libraries.

3.6.5 SPEC CPU2006

We measure HexVASAN's run-time overhead by running the SPEC CPU2006 integer (CINT2006) benchmarks on an Ubuntu 14.04.5 LTS machine with an Intel Xeon E5-2660 CPU and 64 GiB of RAM. We ran each benchmark program on its reference inputs and measured the average run-time over three runs. Figure 3.2 shows the results of these tests. We compiled each benchmark with a vanilla clang/LLVM 3.9.1 compiler and optimization level `-O3` to establish a baseline. We then compiled the benchmarks with our modified clang/LLVM 3.9.1 compiler to generate the HexVASAN results.

The geometric mean overhead in these benchmarks was just 0.45%, indistinguishable from measurement noise. The only individual benchmark result that stands out is that of `libquantum`. This benchmark program performed 880M variadic function calls in a run of just 433 seconds.

3.6.6 Micro-benchmarks

Besides evaluating large benchmarks, we also measure HexVASAN's runtime overhead on a set of micro-benchmarks. We have written test cases for variadic functions with different number of arguments, in which we repeatedly invoke the variadic functions. Table 3.5 shows the comparison between the native and HexVASAN-instrumented micro-benchmarks. Overall,

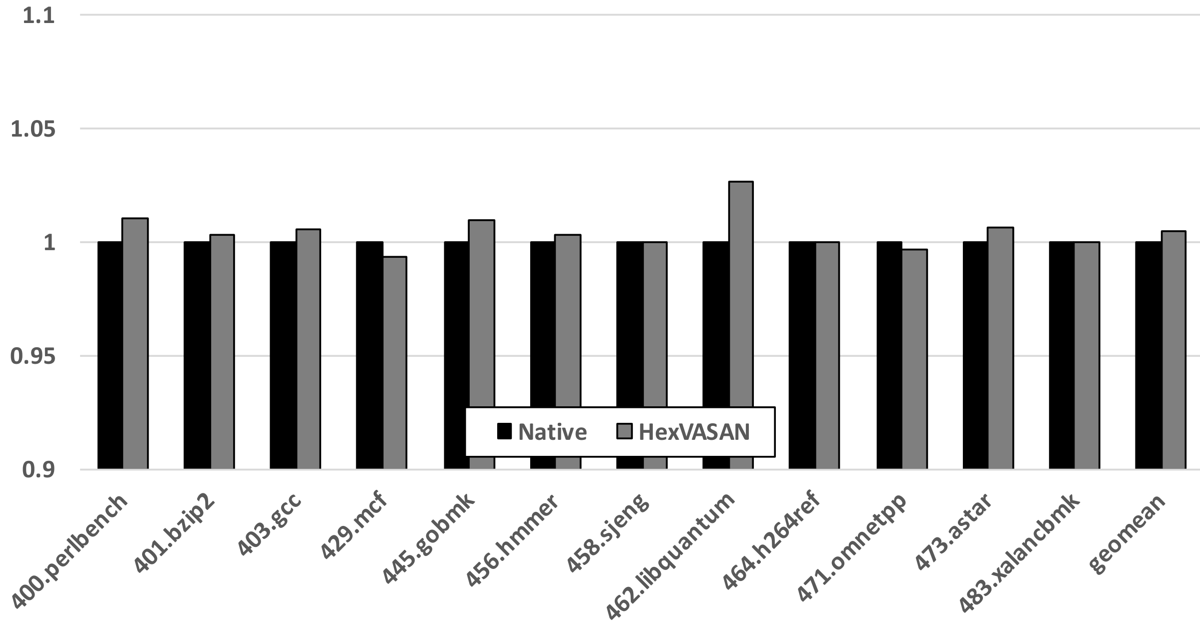

Figure 3.2: Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance.

	# calls	Native [μs]	HexVASAN [μs]
Variadic function argument count: 3	1	0	0
	100	2	12
	1000	20	125
Variadic function argument count: 12	1	0	0
	100	6	22
	1000	55	198

Table 3.5: Performance overhead in micro-benchmarks.

HexVASAN incurs runtime overheads of 4-6x for variadic function calls due to the additional security checks. In real-world programs, however, variadic functions are invoked rarely, so HexVASAN has little impact on the overall runtime performance.

Chapter 4

Moving Beyond Data Space Randomization

4.1 Motivation

Data-only attacks [29, 74, 75, 32] corrupt or leak program data without violating a control flow integrity (CFI) policy. Prior work showed that these attacks are powerful enough to escalate the program's privileges, perform Turing-complete computation, or execute arbitrary system calls, all while bypassing code-reuse defenses such as CFI [55, 43, 29, 74, 75, 122].

To combat data-only attacks, defenders can either retrofit memory-safety techniques such as bounds checking into the program [148], rewrite the program in a safer programming language or language dialect [110, 83, 50], or apply exploit mitigations such as Data-Flow Integrity (DFI) [25], Write Integrity Testing (WIT) [8], or Data Space Randomization (DSR) [15, 20]. Historically, exploit mitigations have been a focus of research, since they usually require less developer effort while also being applicable to legacy software.

Data Space Randomization [15, 20] is a probabilistic exploit mitigation technique that randomizes the *representation* of data in memory to prevent deterministic corruption of data or plaintext information leakage. DSR applies static program analysis to partition all data accesses into disjoint *equivalence classes* based on the aliasing relationships between variables. A pair of variables will be in the same equivalence class if and only if the analysis determines that legitimate data flows could exist between the two.

At run time, the program then randomizes the *representation* of memory objects that fall into different classes. This can be implemented efficiently by using lightweight `XOR` operations with a data-randomization mask (or "key") that is unique to each equivalence class. The compiler generates masks for each data partition and embeds them directly in the binary code of the application [15, 20], or stores them in a dedicated key cache [14]. In both cases, the masks can be protected from memory exploit-based disclosure, either by giving the binary code pages execute-only permissions [38, 13, 59], or by modifying the CPU such that the key cache can only be accessed through custom data access instructions [14].

DSR is only effective as long as the masks remain secret. However, there are a number of possible ways to disclose masks, even if they cannot be read directly. For instance, an adversary could leverage memory vulnerabilities to disclose masked data. If the attacker can choose or inject data (*chosen-plaintext attack*, CPA), knows the data (*known-plaintext attack*, KPA), or guess them with high probability, he can infer the value of a mask used for an equivalence class and all variables in it. Since all variables in an equivalence class have the same mask applied to them, this could then enable further leakage via a JIT-ROP style attack [145]. Alternatively, adversaries could exploit a remote side channel, such as NetSpectre [137], to directly disclose the secret key from the binary or the key cache.

In this chapter, we first demonstrate two novel attacks against DSR. Our attacks disclose DSR masks using direct memory disclosure or side-channel attacks. This allows us to write arbitrary plaintext data into the entire equivalence class whose mask we disclosed. Both

of our attacks are sufficiently generic to work against any prior implementation of DSR, including those that rely on hardware support to protect keys against disclosure, because they exploit design choices in the DSR scheme rather than implementation flaws.

Second, we propose a new data representation rerandomization scheme which thwarts our presented and similar attacks by *dynamically* rerandomizing the masks and updating all program data accordingly. Our defense scheme supports both periodic and on-demand rerandomization. Since purely periodic key refresh could be subject to well-timed attacks, we provide the option to rerandomize at chosen strategic points in the execution of the protected application, thereby ensuring that critical program actions will always trigger rerandomization. To the best of our knowledge, we are the first to present a working rerandomization framework to protect data.

Rerandomizing masks at run time poses unique challenges. For instance, old data representations will no longer be valid under the new mask values. This means, that *all* masked values have to be tracked and updated to ensure correctness. This is non-trivial, since the program might propagate masked values across registers and memory during execution, and rerandomization can trigger at any point in time. To enable reliable rerandomization of data, we add metadata sufficient to identify all masked data and the locations of the masks themselves at any point during program execution.

In summary, our contributions are threefold:

- We present two novel attacks against the current state of the art data defense DSR. Our attacks allow us to exploit attacker-controlled write operations to write arbitrary plaintext data to an encrypted memory location.
- We identify the key challenges towards building an additional layer of moving target defense on top of DSR which rerandomizes the DSR keys and data. We present a new scheme which addresses these challenges. Our prototype, CoDaRR (Continuous

Data Re-Randomization), supports on-demand as well as continuous rerandomization of data. Our evaluation shows that our rerandomization solution is resilient to the attacks against traditional DSR schemes.

- We evaluate the performance of implementation prototype using benchmark suites as well as applications such as *Nginx* and *Thttpd*. Our experiments show acceptable performance in most cases.

4.2 Background

Data Space Randomization DSR thwarts data flows that cannot occur in memory-safe executions of the program by encrypting program variables when they are stored to memory and decrypting them when they are loaded. This way, when the program uses the base pointer of one variable to access a variable in a different equivalence class, the resulting access uses the wrong DR key and yields unpredictable program behavior. An ideal implementation of DSR has the following properties:

1. the DR key used for each memory access should be known a priori and embedded into the binary statically as a read-only constant or an instruction operand;

2. program variables and struct fields should share the same DR key only when they truly alias;

3. the program should randomly generate new masks with high entropy at load time;

4. the program should use strong cryptographic primitives to encrypt memory.

Such an ideal implementation would probabilistically stop most memory corruption attacks, regardless of whether they hijack the control flow of the program or not. Two of the four

```c
int a, b, c;
int * ptr, * ptr2, * ptr3;

if (cond == 1) {
  ptr = &a; ptr2 = &b;
  foo(ptr);
} else {
  ptr = &b; ptr2 = &c;
  foo(ptr3);
}

*ptr = 2;

void foo(int * p) { /* ... */ }
```

Listing 4.1: A simple C program with pointer aliasing

properties (2 and 4) are infeasible in practice, however. To implement property 2, the program needs to know the target object for every memory access instruction, as well as the offset within the target. This requires the compiler to run a fully precise flow-, context-, and field-sensitive pointer analysis, which is undecidable in the general case [88, 131], and does not scale to large programs even when approximated. Implementing property 4 is not practical either because of the high run-time overhead it would incur. Existing implementations of DSR therefore relax both of these properties.

Key Assignment To make DSR practical, existing implementations statically assign masks to memory read and write instructions based on the set of objects they can access. Because most programs use pointers, DSR must take pointer aliasing into account when doing so. Consider, for example, the program shown in Listing 4.1. In this program, variable `ptr` may point to variables `a` or `b`, depending on which path the program executed. The program assigns the value 2 to the target of variable `ptr` and, since both of the possible targets are stored in memory, the value should be encrypted. To allow for encryption with a single, static DR key, DSR assigns the same masks to all variables in `ptr`'s points-to set. All memory accesses to a variable in `ptr`'s points-to set must similarly use the same DR key. This leads

the DSR implementation to merge points-to sets and to construct "equivalence classes" out of pointers with overlapping points-to sets. In the example, pointers `ptr` and `ptr2` are put in the same equivalence class because they can both target variable `b`. Similarly, variables `a`, `b`, and `c` will be encrypted with the same key so they can be accessed from either pointer.

Bhatkar and Sekar's implementation of DSR [15] constructs equivalence classes using Steensgaard's pointer analysis algorithm [149]. This algorithm runs a flow- and context-insensitive pass over all program statements and merges the points-to sets of both pointers whenever one pointer is assigned to another. Formally, the analysis would generate constraint $POINTS_TO(p) = POINTS_TO(q)$ for every assignment statement $p = q$. Cadar et al.'s DSR implementation [20] uses Andersen's pointer analysis [11], which builds subset constraints for all program statements. Under this analysis, an assignment of $p = q$ would generate constraint $POINTS_TO(p) \supseteq POINTS_TO(q)$. Contrary to Bhatkar and Sekar's implementation, Cadar et al.'s implementation computes equivalence classes in a separate step after calculating points-to sets, whereas in the former implementation, every points-to set *is* an equivalence class. Belleville et al.'s HARD uses a context-sensitive version of Steensgaard's analysis [14]. The advantage of this implementation is that it can specialize the call to function `foo` based on its calling context, thereby avoiding the need to collapse `ptr` and `ptr3`'s points-to sets into the same equivalence class because function argument `p` might point to both sets (in different calling contexts).

Lightweight Encryption After constructing the equivalence classes, the DSR scheme inserts the necessary encryption and decryption operations. All existing implementations use XOR masking for this purpose. While XOR masking is not cryptographically secure under key reuse it is more efficient than using symmetric ciphers. Software implementations of DSR embed the XOR keys directly into the binary code of the program and insert metadata to mark the locations of the keys. In this way, the DSR runtime component can randomly

choose keys at program startup and rewrite the binary code in memory. Previous work by Belleville et al. [14] reported an average run-time overhead of 40.96% in fully instrumented programs with context-sensitive masks.

Belleville et al. [14] further presented a hardware implementation of DSR, which stores the masks in a protected memory key table that is backed by a dedicated cache. At run time, the encryption operations look up XOR keys in the key table by their index. Therefore, this implementation is more resilient to direct disclosure attacks, since the XOR keys do not appear anywhere in the binary code of the program. This dedicated hardware support for masked loads and stores reduces the overhead of DSR to 6.61%.

4.3 Attacking DSR

All randomizing mitigations rely on keeping secret information from an adversary. In the case of data space randomization, this secret information is the random XOR key used to mask data in memory. If the attacker can disclose the mask for a targeted class of data, they can undo the effects of DSR and proceed with a traditional memory-corruption attack. In the following motivating attacks against compile-time DSR, we show that we can disclose a mask and bypass DSR by leaking information from the target.

We discuss two different approaches to leaking the secret key and demonstrate that, with a single masked word value from the heap, we can break the guarantees of DSR. First we show a family of attacks that exploits the cryptographic weakness of the XOR masking used in all prior DSR implementations. We then demonstrate a side-channel attack that can disclose keys regardless of the chosen encryption primitive without relying on any properties of the targeted data. To show the practical viability of our attacks, we instantiate them to bypass a DSR protecting the Nginx web server.

4.3.1 Threat Model

The goal of an adversary in this setting is to derandomize and modify DSR-protected data in order to facilitate a data-only exploit, bypassing deployed control-flow hijacking defenses. We note that such multi-step exploits are increasingly popular and common in practice [29, 35, 74, 75, 43, 55, 80].

In our attacks we show how an adversary can disclose information from the protected application in one of two ways: (i) disclosing masked data directly via a memory disclosure vulnerability, or (ii) gradually disclosing data via a side-channel attack such as Spectre [85] or its remote variant NetSpectre [137]. After exploiting one of these disclosure attacks, we assume the attacker can then control a memory-corruption vulnerability sufficient to write to the targeted address once they have computed the required payload.

We target the ideal (theoretical) DSR implementation with perfect alias analysis, i.e., we do *not* rely on implementation weaknesses of any concrete DSR defense. In our direct memory disclosure attacks we assume the DSR implementation uses XOR as the data randomization operation. Although all previous software implementations of DSR rely on XOR masking for performance reasons, our side-channel attack also bypasses implementations that use stronger cryptographic primitives [15, 20, 14].

In summary, we assume the following properties:

- **Vulnerable application:** the target application is a user-space process containing a memory-corruption vulnerability that can overwrite the *targeted data*.
- **Code-injection and code-reuse defenses:** control-flow hijacking attacks are prevented by deployed defenses such as ASLR [123], DEP [104], and CFI [6].
- **Data Space Randomization:** the memory representation of the data the attacker

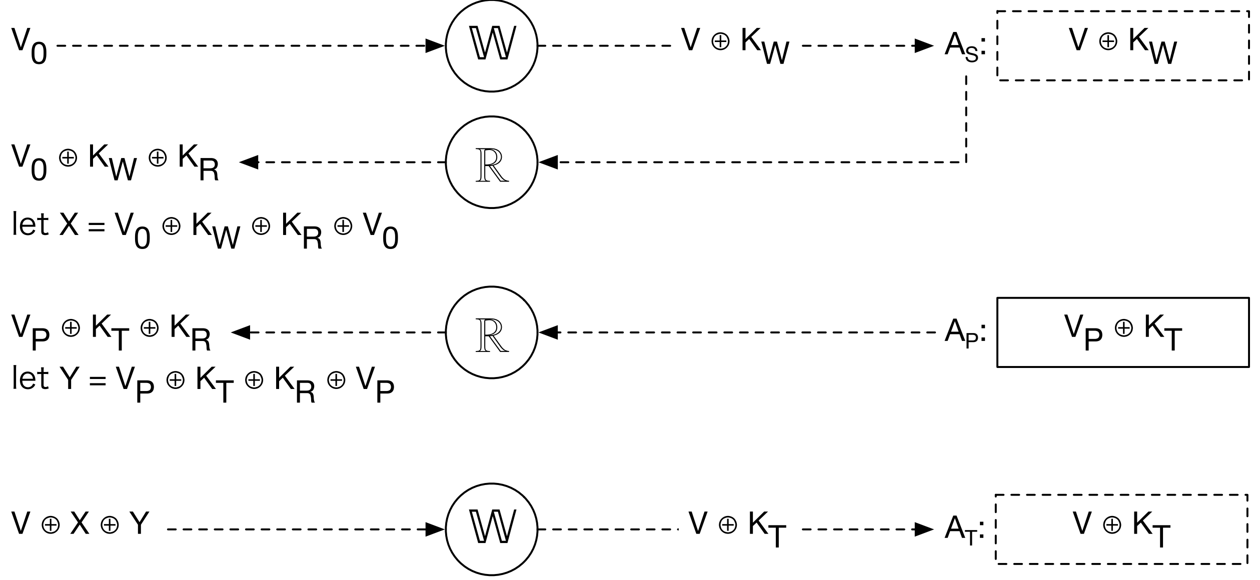

Figure 4.1: Direct memory disclosure attack on a reused XOR key. $\mathbb{R}$: read V, unmasking with key K_R, $\mathbb{W}$: write V, masking with key K_W.

is attempting to modify (*targeted data*) is randomized by DSR. We assume the DSR implementation stores its keys as constants embedded in execute-only code memory so that the keys are not accessible via any direct memory disclosure vulnerabilities.

4.3.2 Attack 1 – Direct Memory Disclosure

XOR masking is not secure when a key is reused to encrypt more than a single value in memory, and we can exploit this property to derandomize data under some circumstances. In Attack 1, we assume that the attacker can find a memory location already containing a known plaintext value masked with the target key, K_T. In practice, attackers should be able to find many such targets because a majority number of variables are initialized with *zero* and more than one variable often share the same key as sound pointer analyses typically overapproximate alias sets [12, 14]. Figure 4.1 illustrates an example attack on reused XOR masking.

Step 1: Derive $X = K_W \oplus K_R$. We first use attacker-controlled write operation $\mathbb{W}$ to write a

known plaintext value V_0 (e.g., zero) to a scratch memory location A_S that will not cause the target process to crash. We then read A_S using attacker-controlled read operation $\mathbb{R}$, disclosing the value $V_0 \oplus K_W \oplus K_R$. Since we know V_0, we can derive $X = K_W \oplus K_R$.

Step 2: Derive $Y = K_T \oplus K_R$. Next, we need to know the plaintext value of a memory location masked with target key K_T. If K_T is only ever used to mask unknown values, this attack will only succeed with the probability that an adversary correctly guesses the relevant bits of one of these plaintext values. Assuming we know the plaintext value V_P stored at A_P but masked with K_T, we can read this value through $\mathbb{R}$ to disclose $V_P \oplus K_T \oplus K_R$, from which we can derive $Y = K_T \oplus K_R$.

Step 3: Overwrite target data. With the values from steps 1 and 2, we have full control of the overwrite of A_T. Assume we want to overwrite A_T with value V. We can write value $V \oplus X \oplus Y$ using $\mathbb{W}$, which results in storing the value $V \oplus K_T$ to A_T, as desired.

For a concrete example of this attack, consider the `ngx_chain_s` struct in Nginx. This linked list struct has two fields, a buffer pointer and a next pointer, which are both masked with the same key. The `next` field of this object is often null, so letting the address of a null `next` field be A_P in the above attack description means that $V_P = 0$ and we can derive the key for the struct, allowing us to rewrite the buffer pointer to a chosen value. While this particular example assumes field-insensitive pointer analysis, the attack still applies as long as the attacker can find a known or controllable encrypted value in the target class.

4.3.3 Attack 2 – Transient Execution

Even without XOR key reuse or a direct memory disclosure, the attacker may still be able to disclose DR keys by exploiting side channel observations. Spectre [85] attacks allow an

adversary to exfiltrate program internals, such as register contents, by causing the target program to speculatively execute an appropriate Spectre gadget. In the following we show how the attacker can leverage this covert channel to disclose DR keys while they are stored in machine registers (*Attack 2.a*) or by speculatively reading execute-only program memory (*Attack 2.b*).

Attack 2.a – Reading a mask from a register with Spectre V2

In this attack we target a DSR mask while it is stored in a register across an indirect call. Listing 4.2 shows an example of code from Nginx that fits the requirements of this attack. The program loads a DR key into register `R15` at label `a:`. It then uses this key, preserves it across the indirect call at `b:`, and reuses the key at `c:`. If an adversary can control the prediction of the call at `b:`, they can cause transient execution of a chosen gadget that leaks some bit(s) from `R15`.

To demonstrate the practical viability of this attack approach, we created a proof-of-concept attack using the BTB-CA-IP Spectre variant [23] targeting the indirect call in Listing 4.2. From an attack process co-located with the target Nginx process, we mistrain the target branch by repeatedly executing an indirect call from the address `b:` to the address where the Spectre gadget is located in the target address space. After sufficient mistraining we send a POST request to the server which triggers the indirect call with attacker control of the third parameter, `size`, which we use to control which bit of the mask the Spectre gadget will leak. We can then observe cache effects of the speculative execution using a timing side channel such as FLUSH+RELOAD [165] or PRIME+PROBE [154].

With the attacking and target processes pinned to the same core of an Intel Core i5-4690K CPU, we were able to successfully leak a 64-bit mask value in about 25 seconds, although further optimization could lower this time. We initially found that our targeted branch

```
ngx_http_do_read_client_request_body:
a: MOV  R15,0x17c3692abe5f0087  ; Load mask into R15
   XOR  RAX,R15                 ; Mask RAX
    ⋮
b: CALL qword ptr [R14 + 0x20]  ; c->recv(c, buf, size);
    ⋮
c: XOR  RAX,R15                 ; Mask RAX
```

Listing 4.2: Target indirect call in Nginx

destination address was almost always cached and could not be reliably mispredicted, so we added a cache flush of this address before the call in Nginx. In a practical setting the attacker could force this value out of cache by evicting its cache set with congruent loads [126, 98]. To simplify our attack implementation we injected a Spectre gadget that leaks a bit of `R15` by loading one of two different array values depending on value of the selected bit. Our attack observes the cache effects of this load using a FLUSH+RELOAD attack with `clflush` and timing instructions we added to the target Nginx process. In practice, an adversary could observe this covert channel from the attacking process or even remotely, as in the NetSpectre attack [137].

Attack 2.b – Reading masks from execute-only memory with Meltdown-PK

As mentioned in Section 4.3.1, we assume masks embedded in program code are protected from direct disclosure by execute-only memory permissions. However, because speculative execution occurs before the CPU checks memory permissions, the attacker may be able to speculatively compute using values from execute-only memory regions and observe side-effects of that computation. Canella et al. demonstrated that they could successfully read values from memory protected by Intel's Memory Protection Keys (MPK) using a Meltdown variant they term *Meltdown-PK* [23]. We speculate that such an adversary may be able to leak masks from a remote target process by using speculative execution to suppress the resulting exception, although we have not explored this idea further.

4.4 Design of a countermeasure

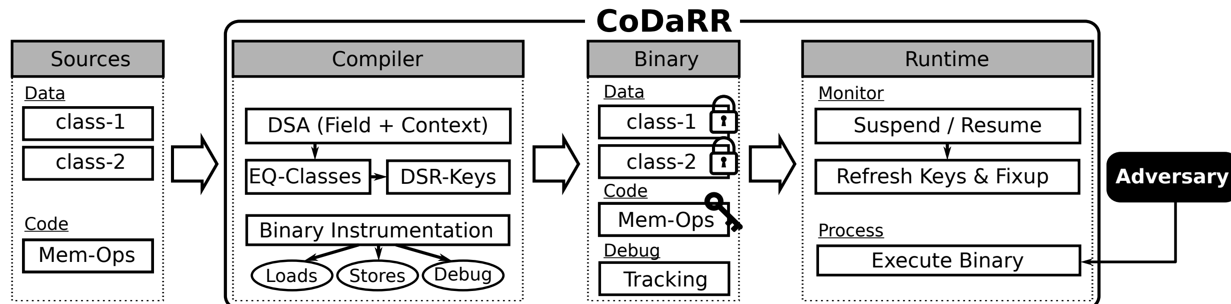

Figure 4.2: Overview of CoDaRR's main components.

As we show in the previous section, all existing DSR implementations are vulnerable to real-world attacks either based on memory corruption or through side channels. We designed our mitigation to be resilient to such attacks. By refreshing the randomization secrets dynamically at run time and updating the program's memory representation accordingly, we ensure that there is not enough time for an adversary to launch an attack after disclosing the necessary secret information. However, since purely periodic rerandomization approaches are prone to timing-window attacks [145, 13], we also incorporate event-driven, on-demand rerandomization. Our mitigation is, to the best of our knowledge, the first to apply rerandomization to the program data representation and could only be realized after we tackled the challenges outlined in the following subsection.

4.4.1 Challenges

Several other exploit mitigations also implement rerandomization, but focus exclusively on code while leaving the data untouched [99, 60, 16, 30, 163, 162, 31]. Designing a rerandomization system for data is fundamentally more difficult than for binary code for the following reasons:

(C-1) Enabling dynamic rerandomization of data. All existing rerandomization approaches focus on relocating and updating code pages since this can be implemented efficiently by routing control-flow transfers through an indirection layer to dynamically adjust code pointers on-the-fly [163, 30, 162]. Applying a similar approach to data transfers would incur excessive run-time overhead, however, as memory access instructions occur much more frequently in real-world programs [9]. As a result, rerandomization of data currently remains an open challenge.

Our primary goal is to enable dynamic rerandomization of data. To achieve this, our solution must refresh masks and re-encrypt all encrypted data accordingly, which means that our solution needs to accurately track all keys and encrypted data. There are several reasons why implementing dynamic key updates is technically challenging: for instance, data accesses under DSR are not atomic operations. A load operation, for example, takes at least two atomic steps, since the load instruction returns a masked value that must be unmasked using a separate instruction. Moreover, updating the keys invalidates all references to the data that still use the old key.

(C-2) Securing the rerandomization mechanism. Our threat model assumes an attacker with the ability to trigger a memory-corruption vulnerability in the victim application. This means that application memory could be disclosed or tampered with unless it is protected by additional defenses. For data randomization specifically, this means the masks must be protected. In the case of static DSR, where all keys are embedded directly into the protected application's code, one could simply apply an execute-only memory mechanism to protect the keys against disclosure attacks [38, 13, 59]. Rerandomization adds a layer of complexity, since the memory that stores the masks must, at least briefly, be writable.

(C-3) Minimizing run-time overhead. In contrast to static Data Randomization, which happens entirely at compile time, our approach incorporates a runtime component to continuously refresh the randomization secrets (i.e., masks). Minimizing any performance overhead introduced by the runtime monitor is another key challenge.

4.4.2 Overview

Figure 4.2 shows the high-level design of our solution and the major components: a compile-time analysis and instrumentation pass, a run-time allocation tracking mechanism embedded into the protected program, and a run-time process monitor that refreshes keys and updates memory.

The compiler first analyzes the source code of the application. Specifically, we compute equivalence classes using a points-to analysis. This is similar to previous DSR schemes. In the figure, the application defines two data structures — `class-1` and `class-2` — which the analysis places into different equivalence classes (see Section 4.4.3).

To provide the information necessary for challenge **C-1**, we extend the compiler to add metadata tracking the locations of masks in registers and spilled to the stack (see Section 4.4.4). The compiler also adds metadata tracking the masks used to encrypt global values, and dynamic instrumentation to track the masks used in the heap (see Section 4.4.5). Combined, the static and dynamic metadata provide our run-time monitor sufficient information to accurately locate and update encrypted values on the heap and in global data as well as DR keys in registers and on the stack.

After compiling the protected application, we launch it along with our runtime monitoring and rerandomization component (see Section 4.4.6). Figure 4.3 illustrates the operation of the monitor at run time. The run-time monitor component runs as a separate process and is

therefore *fully isolated* from the target application. This separation protects randomization secrets and metadata from disclosure by an attacker from the protected process and partially addresses challenge **C-2**. The runtime component pauses the execution of the protected program whenever (i) the program triggers dynamic rerandomization or (ii) the periodic refresh interval expires (Step ①). The monitor is aware of the instrumented code locations that encrypt and decrypt masked values and has fine-grained control over the paused program. The monitor, when pausing the program, can check if there are any in-flight, encrypted values in registers. This can occur in two such cases: either (i) the store instruction following an encryption operation or (ii) the decryption instruction following a load from memory has not yet been executed. The monitor can step forward the execution of the paused program until the pending operation is completed.

When dynamic rerandomization is triggered eventually the monitor refreshes the keys and patches the necessary memory locations and references to use the new keys. It begins by unmapping the old code from memory and mapping in its place a fresh code section with new embedded keys (Step ②). This allows us to minimize the time required to update the code, thus partially addressing challenge **C-3**, while generically supporting fine-grained code rerandomization at the same time. Since the new code may have a different layout, the monitor fixes up all code pointers referring to the old code as detailed in Sections 4.5.1 and 4.5.1. Next, in Step ③ the monitor must update mask values currently stored in registers and on the stack (see Section 4.4.6). Finally, the monitor updates the encrypted data (Steps ④ and ⑤). Using the aforementioned metadata, it locates each memory value that needs to be updated and re-encrypts the value with its new mask.

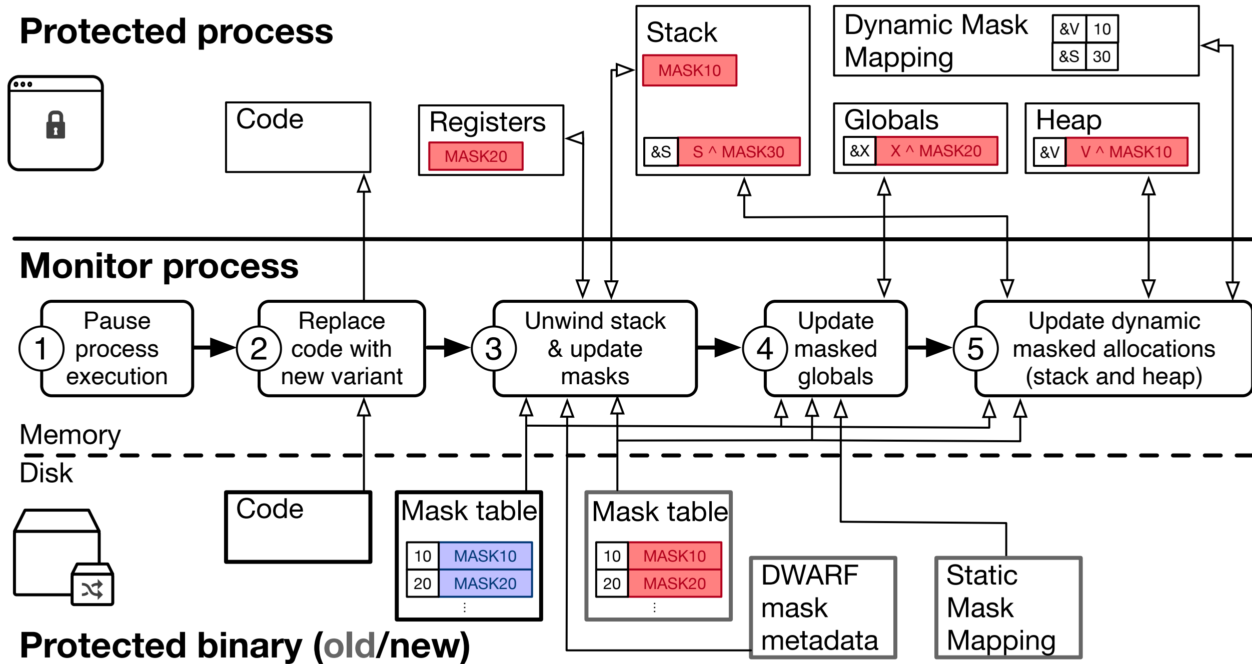

Figure 4.3: CoDaRR run-time rerandomization process.

4.4.3 Compile-time Analysis and Instrumentation

As our first step in preparing a program for data rerandomization, we run Lattner et al.'s [92] Data Structure Analysis (DSA) on the program. DSA is a Steensgaard-style pointer analysis that generates points-to sets for the whole program. We perform context-insensitive analysis using DSA and the equivalence classes our tool computes should therefore be identical to those constructed by Bhatkar and Sekar's DSR [15, 20], which we use as a baseline for comparison.

We then generate a random, word-sized mask value (i.e., 64-bit keys on x86_64 platforms) for each equivalence class and instrument the program at the compiler IR level. We add an XOR operation before all memory stores and after all memory loads with the mask for the corresponding equivalence class. We apply the encryption uniformly to the whole program, and therefore do not exclude "safe" equivalence classes, contrary to prior DSR schemes [15, 20], since these values may still be corrupted by other, unsafe writes.[1] Similar

[1]Using an attack similar to that described in Section 4.3.2, an attacker with arbitrary read and write primitives can derive the mask used for the arbitrary write operation in a straightforward manner.

to prior DSR schemes [14], we only use word-sized keys and use bitwise key shifting to handle unaligned memory accesses.

Note, that our rerandomization mechanism is agnostic to the details of the analysis and instrumentation passes. Our approach could easily support other pointer analyses or stronger forms of encryption.

4.4.4 Mask Table and Tracking Live Masks

After adding the DSR instrumentation, we generate a new metadata section, the Mask Table, containing all of the masks we use in the program, laid out in a contiguous array. The run-time monitor uses this Mask Table as a lookup table (see Section 4.4.6), but it is never loaded into memory by the application itself. The metadata that tracks each value's mask refers to masks by indices in this table to avoid exposing the mask themselves to an attacker.

Mask constants must be loaded into a CPU register before they can be used to encrypt or decrypt data in memory. As in prior work [15, 20, 14], our compiler instrumentation does not prevent mask constants from spilling to the stack, although one could do so, potentially with additional performance and code size overhead. To support rerandomization, our solution therefore needs to keep track of which registers and stack slots currently store masks at the time of rerandomization. One of the challenges to achieving this is to differentiate between actual program values and the keys. To allow for rerandomization at any point of time during program execution, the monitor must know exactly where these masks reside (i.e., in which registers or stack slots) at any given time during the execution of the program.

Instrumentation to dynamically track all mask locations at run time would be slow, since the program must load a mask into a register every encrypted memory operation. To address this problem, we instead emit static metadata that allows the monitor to compute which

registers and stack slots contain keys at each program execution point. Specifically, we create an *artificial* local variable in debug information for every mask constant introduced by DSR, named so that it corresponds to the mask's index in the Mask Table. We then associate each encryption/decryption operation that uses a mask constant with the mask's artificial variable in the program's debug metadata.

The compiler emits DWARF information for mask constants by treating them as local variables, which we force even when compiling at the highest optimization level and with symbol table stripping enabled. This DWARF information contains the symbol name (which references an index in the Mask Table), and describes the location of the associated value throughout the execution of the function.

Note, that the artificial local variables we introduce do not change the runtime behavior or logic of the program in any way, but only affect the debug information generated by the compiler.

As is the case for the Mask Table, the compiler writes the DWARF mask metadata into a file section which is never loaded into memory by the program itself. We only load this section into the runtime monitor's address space. Our DWARF mask metadata is therefore not accessible to an attacker.

4.4.5 Value Mask Mappings

To rerandomize the data representation at run time, our solution must be able to decrypt and re-encrypt all masked data. This requires three steps: locating all masked values, looking up the masks for each value, and re-encrypting all locations with an updated masking key. To support the first two steps, we generate static metadata for global and static local variables, which we call the Static Mask Mapping, and maintain run-time variable metadata for dynamic

allocations on the heap and stack (the Dynamic Mask Mapping). These mappings specify the size, location, and mask of each global variable and heap object, with the mask encoded as an index into the Mask Table.

We generate the Static Mask Mapping at compile time, after we apply the DSR instrumentation and generate the Mask Table. To generate the run-time metadata for dynamic allocations, we hook all memory allocation sites with a small run-time library we embed into the program. Our allocation hook stores the location, size, and mask index of the newly allocated object in the Dynamic Mask Mapping. Our deallocation hook invalidates the metadata.

The Static Mask Mapping is only on located disk and is never loaded into the protected program's address space. The Dynamic Mask Mapping is stored in a shadow memory region in the program's address space, and could therefore be a target for an attacker. However, since the metadata only refers to masks by their index in the Mask Table, disclosure does not reveal the mask values themselves. We refer to Section 4.7.2 for an in-depth discussion on the security of the Dynamic Mask Mapping.

4.4.6 Run-time Monitor

As depicted in Figure 4.3, our run-time monitor is a standalone application that attaches to the protected process and rerandomizes its masks periodically or on demand. When rerandomization is triggered, the monitor first suspends all threads in the program, then updates all masks in the code, in registers, and on the stack. Next, it decrypts all encrypted variables with the old masks and re-encrypts them with the new masks. Finally, it resumes all program threads and repeats this process on set intervals. The protected program can also trigger rerandomization at strategic points during execution.

Updating Code

(Step ②)

To replace masks in the binary code, we replace all code segments in the program with entirely new code segments. Before rerandomization we compile one or more additional variant binaries with different random masks. During rerandomization, we remove the executable code segment of the original binary from memory and map in the executable code segment from a new binary. This approach ensures that we update all masks correctly, without having to mark their locations in the code and minimizes the time required to update the masks for large programs. We do, however, have to account for differences in the code layouts between the original binary and the new binary we mapped in. We do this by updating all code pointers using a strategy laid out in Section 4.5.1.

We could instead use a binary rewriter component to patch the code in-place as some of the existing code rerandomization tools do. Our approach, in our view, is the most generic strategy for DR key replacement as it allows us to use either pre-compiled variants or variants generated at run time by a separate binary rewriter. We discuss this design choice further in Section 4.7.1.

Updating Masks In Registers and Stack

(Step ③)

After replacing the code segment and updating code pointers, we generate call stack information for all threads in the program and identify all active stack frames. Next, we consult the DWARF mask metadata (see Section 4.4.4) to identify all registers and stack slots that currently hold masks. For each mask, we look up its index in the Mask Table of the new variant, and we replace the mask value with the new mask.

Re-Encrypting Global, Stack and Heap Data

(Steps ④ & ⑤)

Next, we iterate through all global variables marked in the Static Mask Mapping (see Section 4.4.5), look up their location, size, and masks, and re-encrypt them with the corresponding mask in the new binary. Also included in the Static Mask Mapping are static local variables, which are actually allocated in the global data section, so we re-encrypt these allocations along with the globals.

Finally, we re-encrypt heap objects and dynamically allocated stack objects by consulting the Dynamic Mask Mapping in the program's shadow memory region.

4.5 Details of our proof-of-concept implementation

We implemented our compile-time analysis and instrumentation passes on top of Clang/LLVM v3.8. We based our static DSR instrumentation on an existing implementation of DSR [132], and modified the instrumentation passes to generate the necessary mask metadata (see Section 4.4.4) and to add calls to our run-time heap metadata management functions (see Section 4.4.5). However, our proposed data rerandomization techniques are not tied to a particular implementation of DSR. We can extend support to other implementations by making similar changes to the compile-time instrumentation passes.

We developed the run-time monitor as a Linux application that attaches to the protected process with the `ptrace` API. In our current implementation, the monitor waits in the background until the rerandomization interval has expired, at which point we attach the monitor to the program and start the rerandomization process. The monitor can also be initiated with chosen rerandomization points (program counter values) in the program to

trigger rerandomization strategically. This latter approach works by setting breakpoints on specific code paths. By hitting a breakpoint, the program generates an exception, which the monitor catches.

4.5.1 Code Pointer Fixups

To minimize the rerandomization overhead and support fine-grained code layout randomization, the monitor rerandomizes masks by replacing the entire code segment of the program with code from a different binary. This means we have to adjust some code pointers after applying rerandomization. Our current implementation does not dynamically track the location of code pointers. We instead use heuristics to find and update a part of the relevant pointers. While these heuristics do not guarantee correctness, similar ones were used in prior work [99], and we found them to be sufficient for the programs we tested with. Prior work [163, 30, 162] has proposed mechanisms to allow more precise updates of code pointers, any of which could be added to our approach.

Updating Return Addresses

After mapping in the new code segments, we start by updating all return addresses to reflect the new code layout. We use the following heuristic to calculate the updated return addresses: first, we disassemble the original binary and variant binaries with new DSR masks using `capstone`. We construct a `ReturnAddressMap` for the original program binary and all variants using Algorithm 3.

Identifying a corresponding return address in the new variants for a given return address in the process becomes trivial once we have `ReturnAddressMap`s for all variants. We prepare this data ahead of time before we attach the monitor to the program for rerandomization.

```
Function CollectReturnAddresses(Binary) is
    declare ReturnAddressMap;
    foreach Function in Binary do
        PreviousIns = nullptr;
        CallNumber=0;
        foreach Ins in Function do
            if IsCallInstruction(PreviousIns) then
                ReturnAddress = Ins.Address;
                ReturnAddressMap.insert(Function, CallNumber++, ReturnAddress);
            end
            PreviousIns=Ins;
        end
    return ReturnAddressMap
end
```

Algorithm 3: Collecting possible return addresses from a disassembled binary.

During rerandomization, we unwind the program stack using `libunwind` and update the return addresses in the stack with new return addresses. This process assumes that equivalent call instructions always appear in the same order in all variant binaries, which holds true even when we apply a software diversity transformation such as function reordering [90]. Basic-block level reordering would require a more sophisticated matching process.

Updating Function Pointers

We first scan the stack, heap, and registers for values in the range of the code section addresses. We replace all pointers we find with the addresses of the corresponding functions in the new binary. This heuristic might misidentify constants as function pointers, but all pointers it identifies correctly will also be updated correctly, as long as the new binary has the same functions. RuntimeASLR implements a similar strategy, but complements it with an offline pointer tracking analysis to distinguish actual function pointer values from constant values [99]. One could add such an analysis to our implementation to improve upon our function pointer updating precision if desired.

4.5.2 Optimizations

One of the main challenges (**C-3**) we set out to meet was to minimize the impact of our rerandomization mechanism on the program's run-time performance. As part of our solution, we chose to update masks by removing the entire code segment of the original program and to map in the code segment from a different pre-compiled binary. On top of this, our monitor does extensive caching of metadata and pre-computes most data structures involved in the rerandomization process. Specifically, our implementation caches the metadata sections containing the Mask Table and Static Mask Mapping (see Sections 4.4.4 and 4.4.5) and the debug metadata sections containing the DSR mask metadata (see Section 4.4.4), for the binary being rerandomized, as well as those of other potential variants. We also pre-compute the `ReturnAddressMap` structures ahead of time (see Section 4.5.1). Finally, to read and write the protected program's memory efficiently, we map the relevant parts of its address space directly into the monitor using the `/proc/pid/mem/` interface.

4.6 Evaluation

We evaluated and analyzed the performance and security of our soultion, CoDaRR, in depth using standard benchmark suites, server applications, as well as real-world application scenarios. For all our experiments, we used an Intel Core i7-8700K machine with 32GB of RAM.

4.6.1 Performance

We evaluated the performance impact of our rerandomization in the context of two popular and widely used web server applications, Thttpd and Nginx, and the impact of our instrumentation

	# Equivalence Classes	Allocation Count	
		Max	Avg
Nginx	886	2,947	7.3
Thttpd	508	569	2.94

Table 4.1: Total number of static equivalence classes and number of allocations per class.

	Globals	Heap	Registers and Stack Spills
Nginx	10,586	4,320	38
Thttpd	4,597	136	16

Table 4.2: Average number of bytes rewritten at dynamic rerandomization time. Globals and Heap columns show bytes re-encrypted; Registers and Stack Spills are in-flight masks replaced.

using standard benchmarks.

For context, Table 4.1 shows the number of equivalence classes as determined by the DSR analysis for each server application, along with the average and maximum number of allocation sites in each class.

Impact of Rerandomization

To assess the performance impact under realistic conditions we tested the version 1.12.0 of the Nginx web server with our rerandomization enabled for different rerandomization intervals. We then ran ApacheBench version 2.3 to generate web traffic, configuring it to send a total of $5,000,000$ web page requests over 200 concurrent network connections, serving a static web page for each of these requests.

The average throughput of the Nginx process due to rerandomization in those experiments is shown in Figure 4.4. As expected, the overall transfer rate (i.e., KB/sec) due to rerandomization drops as rerandomization is triggered more frequently. However, throughput stabilizes between 2.5s and 3s, approaching average baseline DSR throughput (i.e., 40774.81 KB/s

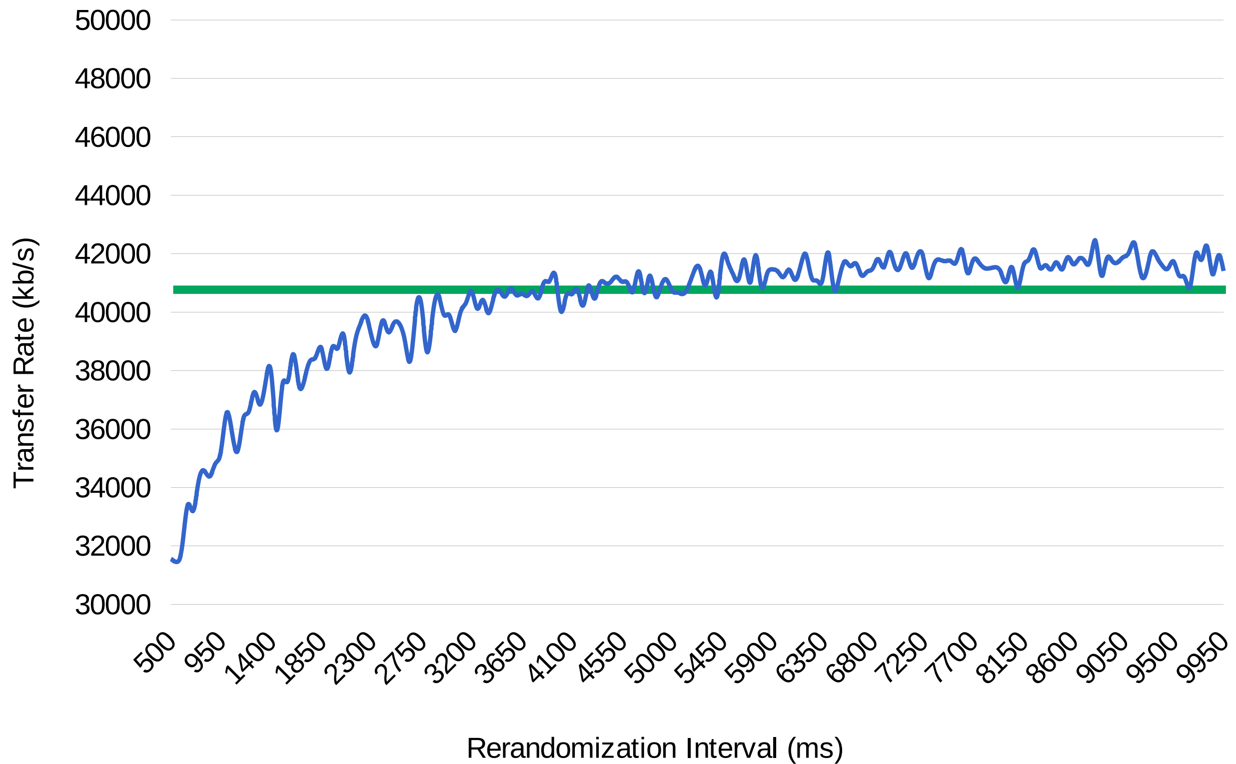

Figure 4.4: Average throughput of Nginx when protected by CoDaRR. We plot the median transfer rate for 5M requests against different rerandomization intervals going from 500ms to 10s. The green line shows throughput for baseline DSR without rerandomization.

depicted as the green line in Figure 4.4). Notably, the throughput degradation due to rerandomization is 3.85% at around 2.3s intervals, the minimum time required to execute a JIT-ROP attack. As we discuss in more detail in Section 4.6.2, the fastest JIT-ROP attack takes 2.3s, while successful BROP and side-channel attacks would take in the order of minutes. This means that our approach provides practical performance at rerandomization intervals that would effectively mitigate these types of attacks. We conducted the same experiment for Thttpd and observed similar results as shown in Figure 4.5. This shows that our implementation can handle even complex server applications such as Nginx and Thttpd, while offering practical performance despite frequent rerandomization. The measurements faster than DSR baseline for long rerandomization intervals in Figures [4.4, 4.5] are simply noisy measurements of throughput.

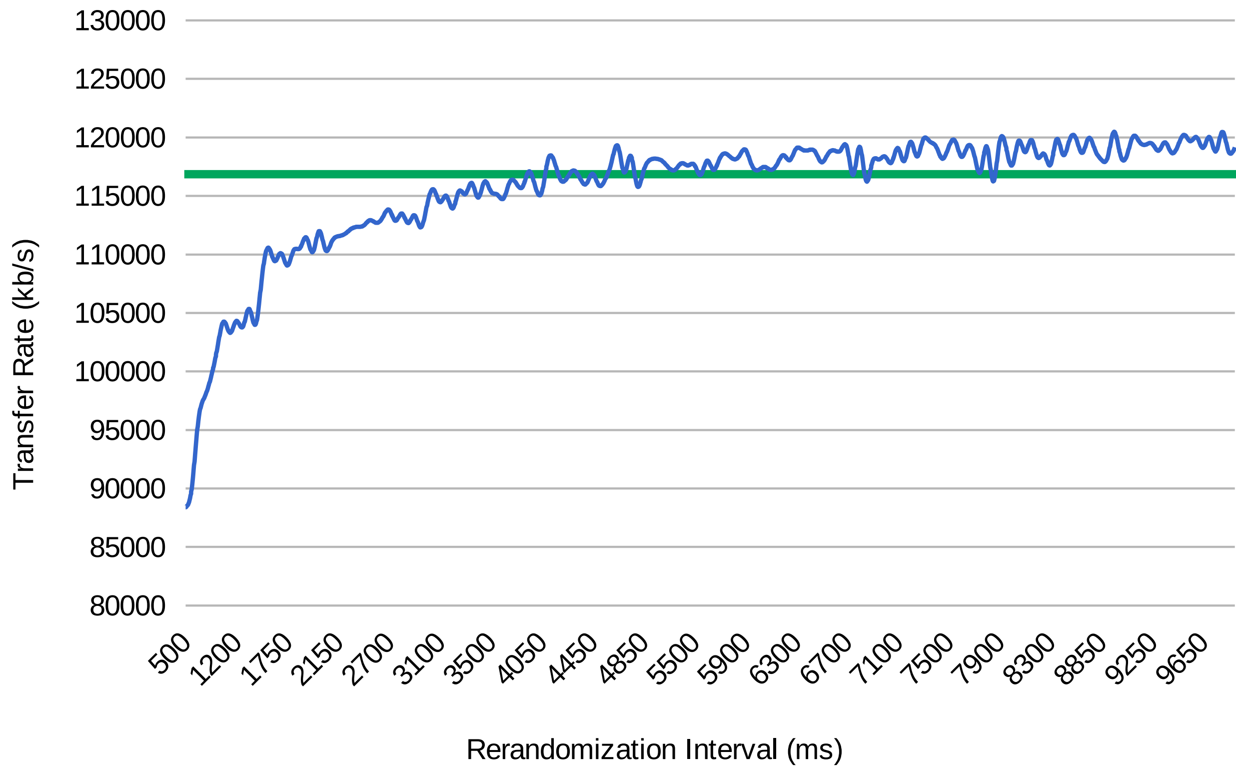

Figure 4.5: Average throughput of Thttpd using the same configuration as in Figure 4.4.

Table 4.2 shows the average amount of data, in bytes, rewritten by the run-time monitor during the rerandomization process in the preceding experiments. Nginx, a larger service, naturally required re-encrypting more data than Thttpd, a minimal service. The *Registers and Stack Spills* column shows the average number of bytes of live mask values that the monitor had to replace in registers or stack slots (see Section 4.4.4). In our experiments we observed no dynamic stack allocated values that needed to be re-encrypted, which is simply a consequence of these particular applications not storing any encryptable objects on the stack at the random program locations where the monitor triggered rerandomization.

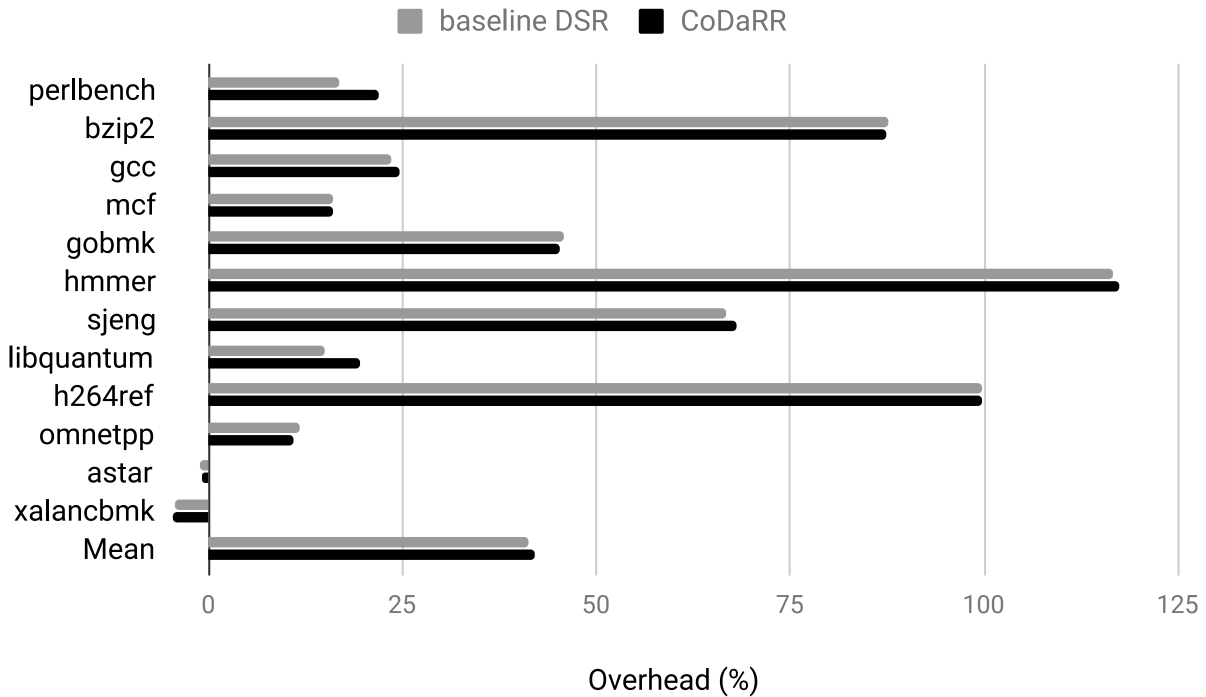

Figure 4.6: SPEC 2006 performance of CoDaRR instrumentation (without rerandomization). Each column shows the median run time of the benchmark with CoDaRR, normalized to the median run time without our tool.

Impact of Dynamic Metadata Tracking

Compared to static DSR approaches, CoDaRR's instrumentation should not add observable performance overhead for running applications since most of our metadata is produced at compile time; this does not affect the application's run-time behavior. The only part that affects the program's run-time behavior is metadata tracking for heap and dynamic local variables. To assess the overall run-time overhead of our dynamic metadata tracking, we instrumented and ran the SPEC CINT 2006 benchmark suite, including all C and C++ benchmarks. We ran all benchmarks three times in their standard configuration on reference inputs, and report CoDaRR's run-time overhead in Figure 4.6. The run-time overhead is normalized to baseline performance (without DSR instrumentation), and depicts the impact of dynamic metadata tracking (i.e., not rerandomization, which we evaluate separately in

Section 4.6.1). The average overhead added by our instrumentation was 0.98% (of the 42.12% total overhead, 41.14% was incurred by the baseline DSR itself [14] without CoDaRR). Although prior DSR implementations did not run SPEC 2006, these results are consistent with the overheads reported in the prior work [14]. Note that, to thoroughly measure the impact of our approach with a large workload, we experimented upon the DSR approach that encrypts all data in memory [14]. Thus, the baseline DSR overhead would be much lower if we base other DSR approaches that encrypt only a limited set of data [15, 20]. Moreover, recent research shows that this DSR instrumentation overhead can be reduced significantly with dedicated hardware support to about 6% [14].

4.6.2 Security Analysis

We now turn towards the security guarantees of our solution. Recall, that in our threat model (Section 4.3.1) we assume a strong adversary with two fundamentally different ways of disclosing data, (i) exploiting a memory-corruption vulnerability or (ii) exploiting a remote side channel such as NetSpectre [137].

Memory Corruption Attacks

By XOR-encrypting all program data with per-equivalence class keys, DSR schemes offer strong protection against adversaries that leak or corrupt data using memory corruption bugs. As we showed in Section 4.3.2, however, DSR does not completely rule out successful memory corruption attacks, since adversaries may be able to infer encryption keys through known plaintext attacks. Inferred keys can then be used to adapt the subsequent exploit and circumvent the defense. This strategy of leaking randomization secrets to adapt the subsequent exploit has been used successfully to circumvent code-centric defenses in the past [17, 145]. These attacks can, however, be very slow. For instance, the fastest JIT-ROP

attack succeeded within 2.3s [145], while a successful BROP attack takes in the order of minutes time (i.e., up to 20 minutes for yaSSL) [17].

Our rerandomization approach counters adaptive data-oriented exploits in two ways. First, we enable explicit rerandomization of the application at run time by exposing an API to application developers. This way, our rerandomization approach can leverage expert knowledge about the program structure, e.g., to enforce rerandomization for security sensitive parts of server applications such as refreshing keys before handling new client connections or upon receiving certain trigger events (e.g., SIGSEGV). From the perspective of the attacker this disrupts any attempts of correlating disclosed information between separate connections. As a result, run time rerandomization mitigates adaptive strategies like BROP which rely on correlation between connections. Second, we additionally employ a continuous rerandomization with an adjustable timing window of at most 2 seconds. This puts a significant time constraint on the attacker, successfully thwarting JIT-ROP style attacks.

Side Channel Attacks

In contrast to memory-corruption based attacks, side channel attacks are limited to information disclosure. This is why, in the context of DSR-based defenses, an adversary has to use the side channel to leak the DSR key data in the first step. We demonstrated one example of such an attack in Section 4.3. We were able to leak a DSR key from a register by mistraining the branch predictor with Spectre V2 [85]. This attack succeeded within 25 seconds, although we believe that further optimization could lower this time somewhat. Still, with our default setting of at least one rerandomization event every 2 seconds, our implementation should be able to defend against Spectre-style attacks.

Furthermore, this attack required a local attacker process pinned to the same physical CPU core as the DSR-protected process. In a more realistic scenario, the attacker would not have

control over a local process, and would have to resort to remote side channel attacks such as NetSpectre instead [137]. NetSpectre leaks bits from the application memory at a much slower rate of about 1 bit per minute. Leaking a single byte of a DSR key would therefore take 8 minutes, which is much longer than our periodic rerandomization interval.

Finally, we would like to point out that (contrary to the protected application) the monitor component does not expose any external interface, and hence, would not be vulnerable to such exploits based on speculative execution.

4.7 Discussion

4.7.1 Generating Variants

For our research prototype, we pre-compiled binaries with randomized DR keys and code layouts for our variants. Alternatively, it would be straightforward to use binary rewriting to produce new variants at runtime as part of the rerandomization process. This rewriter would simply need to know where DR key constants are located in the binary and replace these constants. Previous work [31, 60, 162, 163] has shown binary rewriting approaches can be employed for fine-grained code layout rerandomization.

4.7.2 Attack Against Heap Metadata

In order not to introduce another attack surface, we keep metadata out of the protected process memory as much as possible. We must store heap metadata in memory at run time (Section 4.4.5), and thus we need to consider the security implications of this in-process metadata. The attacker may try to leak the content of this metadata which would allow an adversary to read heap object addresses and the associated indices into the Data

Randomization key table. Using this information, the attacker could infer what heap allocations are in the same equivalent class. Knowing these indices, however, will not allow the attacker to learn the actual DR keys that used to encrypt the heap data because the key table itself is never loaded into process memory. Recall that we assume DR keys used in masking operations are embedded as constants into code memory without read permissions.

An active attacker may attempt to corrupt this in-memory metadata to manipulate the Monitor's heap data re-encryption process. Corrupting an address entry of this metadata, for example, the attacker may trick our run-time monitor into re-encrypting unintended heap data, leaving the original data unchanged after rerandomization. However, this still does not allow the attacker to manipulate the program in the intended way because after rerandomization, the program will decrypt this data with a new encryption key. This incorrect decryption of data will lead to incorrect, unpredictable program behavior, since the correct plaintext value cannot be recovered.

Chapter 5

Related Work

5.1 Dynamic Bug Finding – Sanitizers

Static tools analyze program source code and produce results that are conservatively correct for all possible executions of the code [34, 61, 138, 22, 21]. In contrast, dynamic bug finding tools—often called "sanitizers"—analyze a single program execution and output a precise analysis result valid for a single run only. The biggest difference between sanitizers and mitigation solutions is the type of security policy they enforce. Exploit mitigations deploy a policy aimed at detecting or preventing attacks, whereas sanitizers aim to pinpoint the precise locations of buggy program statements. Many sanitizers are now available and some have seen widespread adoption. They play a critical role in finding security issues in programs during pre-release testing.

Location-based access checkers can use red-zone insertion [69, 141, 19, 68, 139] or guard pages [127, 105] to detect spatial safety violations. Either of these techniques can be combined with reuse delay to additionally detect temporal safety violations [69, 84, 141, 19, 139, 127, 105, 48, 41]. AddressSanitizer [139] (ASan), instruments memory accesses and allocation

sites to detect spatial memory errors, such as out-of-bounds accesses, as well as temporal memory errors, such as use-after-free bugs.

Undefined Behavior Sanitizer [47] (UBSan) instruments various types of instructions to detect operations whose semantics are not strictly defined by the C and C++ standards, e.g., increments that cause signed integers to overflow, or null-pointer dereferences. Thread Sanitizer [140] (TSAN) instruments memory accesses and atomic operations to detect data races, deadlocks, and various misuses of synchronization primitives. Memory Sanitizer [150] (MSAN) detects uses of uninitialized memory.

Pointer casting monitors detect illegal downcasts through the C++ `static_cast` operator. Illegal downcasts occur when the target type of the cast is not equal to the run-time type (or one of its ancestor types) of the source object. UndefinedBehaviorSanitizer [47] (UBSan) and Clang CFI [46] include checkers that verify the correctness of `static_cast` operations by comparing the target type to the run-time type information (RTTI) associated with the source object. This effectively turns `static_cast` operations into `dynamic_cast`. The downside is that RTTI-based tools cannot verify casts between non-polymorphic types that lack RTTI.

CaVer [93] targets verifying correctness of downcasts in C++. Downcasting converts a base class pointer to a derived class pointer. This operation may be unsafe as it cannot be statically determined, in general, if the pointed-to object is of the derived class type. TypeSan [67] is a refinement of CaVer that reduces overhead and improves the sanitizer coverage.

UniSan [100] sanitizes information leaks from the kernel. It ensures that data is initialized before leaving the kernel, preventing reads of uninitialized memory.

5.2 Variadic Function Attack Surface Mitigations

To address the variadic function attack surface, we presented a comprehensive solution which can either be used as an always-on runtime monitor to mitigate exploits or as a sanitizer to detect bugs, sharing similarities with the sanitizers 5.1 that exist primarily in the LLVM compiler.

Different control-flow hijacking mitigations offer partial protection against variadic function attacks by preventing adversaries from calling variadic functions through control-flow edges that do not appear in legitimate executions of the program. Among these mitigations, we find Code Pointer Integrity (CPI) [87], a mitigation that prevents attackers from overwriting code pointers in the program, and various implementations of Control-Flow Integrity (CFI), a technique that does not prevent code pointer overwrites, but rather verifies the integrity of control-flow transfers in the program [6, 161, 42, 114, 121, 128, 169, 170, 33, 40, 81, 57, 44, 116, 153, 117, 102, 107, 129, 168, 156, 167, 106, 18, 58, 115, 124].

Control-flow hijacking mitigations *cannot* prevent attackers from overwriting variadic arguments directly. At best, they can prevent variadic functions from being called through control-flow edges that do not appear in legitimate executions of the program. We therefore argue that our approach and these mitigations are orthogonal. Moreover, prior research has shown that many of the aforementioned implementations fail to fully prevent control-flow hijacking as they are too imprecise [62, 45, 24, 52], too limited in scope [135, 146], vulnerable to information leakage attacks [51], or vulnerable to spraying attacks [119, 63]. We further showed in Section 3.6.1 that variadic functions exacerbate CFI's imprecision problems, allowing additional leeway for adversaries to attack variadic functions.

Defenses that protect against direct overwrites or misuse of variadic arguments have thus far only focused on format string attacks, which are a subset of the possible attacks on variadic functions. LibSafe detects potentially dangerous calls to known format string functions such

as `printf` and `sprintf` [155]. A call is considered dangerous if a `%n` specifier is used to overwrite the frame pointer or return address, or if the argument list for the `printf` function is not contained within a single stack frame. FormatGuard [36] instruments calls to `printf` and checks if the number of arguments passed to `printf` matches the number of format specifiers used in the format string.

Shankar et al. proposed to use static taint analysis to detect calls to format string functions where the format string originates from an untrustworthy source [144]. This approach was later refined by Chen and Wagner [27] and used to analyze thousands of packages in the Debian 3.1 Linux distribution. TaintCheck [112] also detects untrustworthy format strings, but relies on dynamic taint analysis to do so.

`_FORTIFY_SOURCE` of *glibc* provides some lightweight checks to ensure all the arguments are consumed. However, it can be bypassed [2] and does not check for type-mismatch. Hence, none of these aforementioned solutions provide comprehensive protection against variadic argument overwrites or misuse.

5.3 Security Mitigations

Given the lack of memory safety guarantees in C and C++ programs, we must operate under the assumption that the programs written in these languages will have memory corruption errors that can be exploited. Mitigations against memory corruption exploits can use security enforcement policies typically based on static analysis of source or binary code. The enforcement based mitigations may also collect information about the execution of the program to fortify the metadata necessary to support the enforcement policies. The second type of mitigations are randomization based mitigations which depend upon randomizing the low level details of the process such as code layout or data representation of the program.

Randomization based protections offer probabilistic security guarantees; more fine-grained approaches are harder to implement but offer greater entropy.

5.3.1 Control-Flow Exploit Defenses

Data Execution Prevention (DEP) realized by W⊕X feature in operating systems is a memory protection mechanism that makes every page in the program memory either writable or executable. DEP essentially eliminates traditional code injection attacks as the code injected to data sections will not be executable.

Control Flow Integrity (CFI) enforces at runtime that the control flow targets of indirect control flow transfer instructions (iCFT) belong to a pre-determined set of targets for the given iCFT instruction. iCFTs include indirect branches and return instructions. Indirect branches are also referred to as forward edge control-flow instructions and CFI can employ static source analysis to derive a conservative set of targets for a given indirect branch instruction at compile time. And for the backward edge control-flow transfers originating from return instructions additional dynamic information is collected to pair backward edges with the most recent caller.

Backward edge protection offered by CFI methods are too permissive; Shadow stack mitigation protects the return addresses by maintaining a copy of the function return addresses in a disjoint protected stack region termed shadow stack region. When return instructions are executed, the return address in stack memory is compared against return address stored in shadow stack to enforce backward edge control flow integrity. DEP, CFI and Shadow stack mitigations together can stop most code injection, code reuse and Return Oriented Programming (ROP) exploits through their enforcement policies. DEP is supported by major operating systems. CFI and shadow stack are available in popular compiler toolchains such as clang/LLVM and GCC and are widely deployed to protect against control-flow attacks.

Address Space Layout Randomization (ASLR) randomizes the base addresses of the code, heap, stack and global memory sections in program memory. ASLR is adopted widely for both operating system kernels and user space applications. Any code reuse exploit must compute the location of the target gadget location to complete a successful attack. Randomizing the base offsets increases the difficulty of this computation. ASLR is a coarse-grained randomization solution which can be bypassed by adaptive exploits which first identify a pointer in memory and de-randomize the target memory regions based on the knowledge derived from the initial step.

To improve the entropy of protection against control-flow attacks more fine-grained randomization solutions were proposed; profile guided NOP insertion [73], randomizing all instructions in program memory [71], randomization of code generated by just-in-time (JIT) compilers [72] and other solutions improved the state of the art randomization mitigations against code reuse attacks.

Isomeron [37] randomizes the code layout by switching between two diversified code layouts after every function call and return instruction execution. Since the gadget locations differ between the two code layouts, Isomeron probabilistically breaks the chained execution of control-flow gadgets.

Counterfeit object-oriented programming (COOP) attacks [136] exploit the virtual dispatch mechanism used by C++. This approach invokes attacker chosen virtual methods on attacker injected objects and bypasses coarse-grained CFI mitigation. Readactor++ [39] employs randomization of C++ virtual tables as well as booby-trapping to stop COOP attacks. Subversive-C [95], another COOP attack exploits Objective-C class metadata and cannot be protected by virtual table randomization since the Objective-C class metadata is mutable. To mitigate attacks against Objective-C runtime, Lettner et. al [95] added a message authentication code (MAC) to every sensitive object. Their mitigation validates the integrity of the runtime data structures before control-flow transfers that indirectly use the contents of

the protected data structures.

Crane et al. [37] proposed a dynamic randomization scheme to prevent side-channel based information leak attacks. They create multiple variants of program functions and apply different diversifying transformations to each variant. The diversified layout introduces randomized noise to observable leakage in the shared cache which improves the entropy against side-channel attacks.

5.3.2 Data-Oriented Exploit Defenses

As the defenses against control-flow attacks evolved over time data-oriented have become an attractive alternative for system compromise. Similar to control-flow mitigations the data-oriented defenses had seen both enforcement and randomization based solutions. Data-oriented attacks modify the program behavior without violating the control-flow of the program to bypass control-flow defenses.

Write Integrity Testing (WIT) [8] enforces that the target of a memory write is in the set of allowed objects. The policy is based on static alias analysis which determines the set of objects that each write instruction is allowed to modify. Each write instruction and associated objects are assigned a unique color; the enforcement policy makes sure that only the write instructions with the correct color identifier is used to write to the target object during execution. A color table is maintained to track object allocations, and the write operations are checked against the color table and the associated static metadata.

Data Flow Integrity (DFI) [25] is another enforcement based mitigation against data-oriented attacks. DFI tracks both write and read operations compared to WIT which tracks only the write operations. DFI at run time intercepts each read instruction in the program and verifies that the data read from memory was not corrupted by illegal data flows in the program. DFI

pre-computes at compile time which write instructions can write to the memory read by each read instruction. This static metadata is associated with memory locations to which writes are performed at runtime to enforce DFI policy. Since DFI protection is applied to all memory locations, DFI can also stop code reuse attacks that attempt to modify instruction pointers and return addresses.

The static metadata for both WIT and DFI are generated by computing the data-flow graph of the program at compile time. Computing data-flow graph at compile time is an undecidable problem in the general case [88, 131], and does not scale to large programs even when approximated resulting in a conservative analysis. For DFI enforcement, this means some of the illegal data flows cannot be determined. WIT and DFI enforcements therefore suffer due to the lack of precision in the underlying static analysis. Moreover, DFI adds security checks for both read and write operations and pure software implementations of inter-procedural DFI schemes incur a prohibitive runtime overhead of 103%. The hardware implementation of DFI (HDFI) incurs a lower overhead but the number of protection domains offered by HDFI is limited.

Bhatkar and Sekar [15], and Cadar et al. [20] proposed Data Space Randomization (DSR) independently and concurrently. The static analysis used by DSR places pointers that alias under equivalence classes; two variables are placed under the same equivalence class if the analysis determines there can be a legitimate data flow between the two variables. Each equivalence class is allocated an encryption key which is then used to encrypt all the data of the equivalence class before storing them in program memory. When a variable that belongs to one equivalence class is used to read or write data that belongs to a different equivalence class due to illegal data flows, these operations will result in usage of wrong encryption key for the data targeted by the operation. The protection offered by DSR like WIT and DFI relies on the precision of the underlying static analysis but the reported runtime overhead of 40% is better than that of DFI. The hardware assisted DSR [14] brings the overhead around

6.61% and offers improved protection against direct disclosure attacks as the encryption keys are protected using hardware support.

5.4 Rerandomization Defenses

Rerandomization has been proposed in several prior works, but has so far only been applied to the protected program's memory layout and to pointers [99, 60, 16, 30, 163, 162, 31, 56]. These techniques rerandomize the memory layout of the protected application and update all references that exist in the program accordingly. Whereas code references in executable code are easy to identify and update, code references in data sections are not. OS-level ASR [60] and TASR [16] therefore modify the compiler to emit location information for code pointers in data sections.

Remix only applies intra-function randomization (i.e., basic block shuffling without crossing function boundaries), which removes the need for fine-grained pointer tracking [31]. Shuffler [163], CodeArmor [30], and ReRanz [162] route control-flow transfers through an indirection mechanism which either simplifies the code pointer updating problem by isolating code pointers from the rest of the program's address space [30, 163], or by enabling on-the-fly pointer adjustments [162]. RuntimeASLR uses dynamic instrumentation tools to generate pointer tracking policies [99]. Morpheus uses memory tagging and a customized RISC-V processor core to tag code, data, and pointers thereto. Morpheus relies on these tags to periodically rerandomize and re-encrypt pointers in hardware [56]. OS-level ASR [60] and TASR [16] are the most similar to CoDaRR since we also modify the compiler to generate location information for our masks.

Code layout rerandomization solutions cannot completely protect against data-oriented attacks. The data-oriented attacks can operate completely on the data plane and Hu et

al. [75] showed that even certain types of code reuse attacks can be constructed with the help of Data Oriented Programming which cannot be detected by TASR. Shapeshifter [160] employs rerandomization of the data layout of “whitelisted” security-critical data objects and variables. Their rerandomization scheme triggers randomization based on number of accesses to the “whitelisted” objects. This means, their security policy only applies to (and protects) this subset of the data plane.

As we noted in Chapter 4, all DSR protections are vulnerable against transient execution based adaptive attack strategies and known plaintext attacks. Unlike code layout rerandomization solutions, there are no data representation rerandomization solutions in the literature. To the best of our knowledge, our solution CoDaRR presented by us in Chapter 4 is the first solution to perform representation rerandomization. In contrast to the existing work, CoDaRR continuously rerandomizes the DSR masks and updates *all* program data accordingly

Chapter 6

Conclusions

Traditional memory corruption errors, despite the myriad solutions developed over the years towards identifying them, still offer enough attack surface to launch exploits. Data-oriented exploits circumvent control-flow mitigations while the current generation of randomization based data attack defenses are vulnerable to direct memory disclosure and transient execution attacks.

Variadic functions are prevalent in popular programs (Firefox, Apache, CPython, Nginx, Wireshark) and libraries frequently use variadic functions to offer flexibility. Current solutions, including static type checkers and CFI implementations, do not find variadic argument type errors or prevent attackers from exploiting calls to variadic functions. The lack of type safety of variadic function arguments is the root cause of these exploits and we have developed a comprehensive approach to address this problem. Our approach effectively type checks variadic arguments at run time, imposes a negligible overhead (0.45%) on the SPEC CPU2006 benchmarks and is an effective mitigation against variadic function exploits.

In our work, we investigated the security capabilities of Data Space Randomization (DSR). DSR diversifies the memory representation of applications by *masking* data values with

randomly chosen keys. Our attacks against DSR show that all existing DSR approaches can be bypassed. In our first attack, we corrupted deterministically memory locations protected by DSR using memory corruption. Our second attack showed that through transient execution we can leak DSR keys paving way for adaptive multi-stage attacks that can bypass DSR.

In response to our findings, we designed a moving target defense that rerandomizes the representation of program data. Our approach accounts for both traditional memory corruption exploits as well as the evolving landscape of transient execution attacks. We presented the design of a novel DSR scheme with a fully dynamic key schedule. Our approach is to the best of our knowledge, the first scheme to enable rerandomization of program data. Our mitigation continuously rerandomizes protected applications at periodic randomization intervals, as well as on-demand using event-based triggers. Our evaluation confirms that our approach is practical, incurring limited run-time overhead in standard benchmarks, and is able to handle complex, real-world applications, such as Nginx and Thttpd. Furthermore, our solution is resilient against information disclosure attacks, such as our presented attacks based on memory corruption and speculative execution.

Bibliography

[1] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617.

[2] A eulogy for format strings. http://phrack.org/issues/67/9.html.

[3] Kraken benchmark. https://wiki.mozilla.org/Kraken.

[4] Octane benchmark. https://developers.google.com/octane/faq.

[5] Using the gnu compiler collection (gcc) - function attributes. https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. volume 13, Nov. 2009.

[7] M. Abadi, M. Budiu, Ú. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI, 2006.

[8] 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, May 2008.

[9] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter. X86-64 instruction usage among c/c++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 68–79, 2019.

[10] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[11] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[12] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, pages 332–341. ACM, 2005.

[13] 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*, CCS '14, page 1342–1353, 2014.

[14] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, and M. Franz. Hardware assisted randomization of data. In M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 337–358, 2018.

[15] S. Bhatkar and R. Sekar. Data space randomization. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, 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*, CCS '15, page 268–279, 2015.

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

[18] D. Bounov, R. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[19] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO)*, 2011.

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

[21] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[22] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[23] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[24] 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*, 2015.

[25] 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, 2006.

[26] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007.

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

[29] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, pages 177–192, 2005.

[30] 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, April 2017.

[31] 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*, CODASPY '16, page 50–61, 2016.

[32] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 114–128. IEEE, 2019.

[33] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[34] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[35] 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*, CCS '15, page 952–963, 2015.

[36] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[37] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[38] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. 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, May 2015.

[39] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[40] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[41] T. H. Dang, P. Maniatis, and D. Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security Symposium*, 2017.

[42] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[43] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[44] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference (DAC)*, 2014.

[45] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[46] L. Developers. Control flow integrity. `https://clang.llvm.org/docs/ControlFlowIntegrity.html`, 2017.

[47] L. Developers. Undefined behavior sanitizer. `http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2017.

[48] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2006.

[49] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *International Conference on Software Engineering (ICSE)*, 2012.

[50] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked c: Making c safe by extension. In *IEEE Cybersecurity Development (SecDev)*, 2018.

[51] I. Evans, S. Fingeret, J. González, 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[52] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[53] Exploit Database. sudo_debug privilege escalation. `https://www.exploit-db.com/exploits/25134/`, 2013.

[54] A. Francillon and C. Castelluccia. Code injection attacks on Harvard-architecture devices. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[55] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: Hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2405–2419, 2017.

[56] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, and et al. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 469–484, 2019.

[57] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[58] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symp. on Security and Privacy*, 2016.

[59] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, page 325–336, 2015.

[60] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, 2012.

[61] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[62] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[63] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX Security Symposium*, 2016.

[64] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[65] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh. Page cache attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 167–180, New York, NY, USA, 2019. Association for Computing Machinery.

[66] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.

[67] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. Typesan: Practical type confusion detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[68] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[69] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter Conference*, 1992.

[70] Hex-Rays. Ida pro.
`https://www.hex-rays.com/products/ida/`, accessed April 30, 2020.

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

[72] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[73] 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, 2013.

[74] 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, Aug. 2015.

[75] 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, May 2016.

[76] Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Dec. 2014.

[77] Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, Dec. 2011.

[78] ISO/IEC JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming Languages — C, 2011.

[79] ISO/IEC JTC1/SC22/WG14. ISO/IEC 14882:2014, Programming Language C++, 2014.

[80] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[81] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[82] J. Jelinek. FORTIFY_SOURCE. `https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html`, 2004.

[83] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[84] R. W. Jones and P. H. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[85] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[86] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

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

[88] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1991.

[89] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[90] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[91] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004.

[92] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[93] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, 2015.

[94] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[95] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, M. Franz, and U. Irvine. Subversive-c: Abusing and protecting dynamic message dispatch. In *USENIX Annual Technical Conference*, 2016.

[96] Linux Programmer's Manual. va_start (3) - Linux Manual Page.

[97] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard. Armageddon: Last-level cache attacks on mobile devices. *ArXiv*, abs/1511.04897, 2015.

[98] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[99] K. Lu, W. Lee, S. Nürnberger, and M. Backes. How to make aslr win the clone wars: Runtime re-randomization. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[100] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[101] 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 *SOSP '11*, 2011.

[102] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[103] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0.99*, 2013.

[104] Microsoft. Data execution prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US`, 2006.

[105] Microsoft Corporation. How to use pageheap.exe in windows xp, windows 2000, and windows server 2003, 2000.

[106] Microsoft Corporation. Control Flow Guard (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`, 2016.

[107] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[108] 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, 2018.

[109] National Vulnerability Database. NVD - CVE-2009-1897. `https://nvd.nist.gov/vuln/detail/CVE-2009-1897`, 2009.

[110] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37. ACM, 2002.

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

[112] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[113] R. NISSIL. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886`.

[114] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[115] B. Niu and G. Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[116] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[117] B. Niu and G. Tan. Per-input control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[118] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi. Hardscope: Thwarting dop with hardware-assisted run-time scope enforcement. *ArXiv*, abs/1705.10295, 2017.

[119] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX Security Symposium*, 2016.

[120] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[121] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[122] T. Park, J. Lettner, Y. Na, S. Volckaert, and M. Franz. Bytecode corruption attacks are real—and how to defend against them. In C. Giuffrida, S. Bardin, and G. Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 326–348, 2018.

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

[124] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[125] C. Percival. Cache missing for fun and profit. 2005.

[126] C. Percival. Cache missing for fun and profit, 2005.

[127] B. Perens. Electric fence malloc debugger, 1993.

[128] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[129] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[130] Radare org. Radare.
https://www.radare.org/r/, accessed April 30, 2020.

[131] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[132] rinon. Multicompiler datarando. https://github.com/securesystemslab/multicompiler/tree/master/lib/DataRando, 2018.

[133] 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, 2012.

[134] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. G. Zorn. Modular protections against non-control data attacks. *2011 IEEE 24th Computer Security Foundations Symposium*, pages 131–145, 2011.

[135] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[136] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[137] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 279–299, 2019.

[138] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[139] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[140] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009.

[141] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[142] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[143] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[144] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[145] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[146] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[147] Solar Designer. Getting around non-executable stack (and fix). Email to the Bugtraq mailing list, Aug. 1997.

[148] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[149] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[150] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[151] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *European Workshop on System Security (EuroSec)*, 2009.

[152] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[153] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[154] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 2010.

[155] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. *White paper, Avaya Labs*, 2001.

[156] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[157] V. Van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2012.

[158] Vector 35 Inc. binary.ninja : a reversing engineering platform. `https://binary.ninja/`, accessed April 30, 2020.

[159] 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*, SOSP '93, page 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[160] Y. Wang, Q. Li, Z. Chen, P. Zhang, and G. Zhang. Shapeshifter: Intelligence-driven data plane randomization resilient to data-oriented programming attacks. *Computers & Security*, page 101679, 2019.

[161] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[162] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In *International Conference on Virtual Execution Environments (VEE)*, 2017.

[163] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[164] J. Xiao, H. Huang, and H. Wang. Kernel data attack is a realistic security threat. In B. Thuraisingham, X. Wang, and V. Yegneswaran, editors, *Security and Privacy in Communication Networks*, pages 135–154, Cham, 2015. Springer International Publishing.

[165] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[166] Y. Yarom and K. E. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2013:448, 2013.

[167] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[168] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[169] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[170] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.