UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Finding and Mitigating Memory Corruption Errors in Systems Software

Permalink
https://escholarship.org/uc/item/72p5c8nv

Author
Lettner, Julian

Publication Date
2018

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Finding and Mitigating Memory Corruption Errors in Systems Software

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Julian Lettner

Dissertation Committee:
Professor Michael Franz, Chair
Professor Brian Demsky
Professor Sam Malek

2018

Chapter 3 and 5 © 2016 USENIX.

Chapter 4 and 5 © 2018 Springer.

All other materials © 2018 Julian Lettner.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Prof. Michael Franz. Over the past five years, he has been an incredible mentor who has provided guidance and encouragement throughout my studies at UCI. Prof. Franz welcomed me into his Secure Systems and Software lab where I had the opportunity to collaborate and work with an amazing group of researchers and friends.

I would like to thank my postdocs, Dr. Per Larsen, Dr. Stijn Volckaert, and Dr. Yeoul Na, for their contributions, ideas, and advice that have facilitated my research efforts. I am especially grateful to lab alumni, Andrei Homescu and Stephen Crane, for their invaluable support and willingness to share their expertise. In addition, I would like to thank my friends in the lab for being great colleagues and for making graduate school much more enjoyable. Brian, Taemin, Prabhu, Anil, Alex, Mohaned, Joseph, Dokyung, and Paul—it has been a pleasure working with you!

Finally, I would like to give special thanks to my committee members, Prof. Brian Demsky, Prof. Sam Malek, and Prof. Daniel Gillen for their time and consideration.

Portions of this dissertation have been previously published in conference proceedings. To my coauthors on these projects, thank you for your contributions on these publications.

Chapter 3 and portions of Chapter 5 are reprinted, with permission, from Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. *Subversive-C: Abusing and Protecting Dynamic Message Dispatch.* In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16). 2016.

Chapter 4 and portions of Chapter 5 are reprinted, with permission, from Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. *PartiSan: Fast and Flexible Sanitization via Run-time Partitioning.* In Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID '18). 2018.

This research is based on work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, the National Science Foundation under awards CNS-1619211 and CNS-1513837, the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, as well as gifts from Mozilla, Oracle, and Qualcomm.

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), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

CURRICULUM VITAE

Julian Lettner

EDUCATION

Doctor of Philosophy in Computer Science — **2018**
University of California, Irvine — *Irvine, California*

Master of Science in Software Engineering — **2012**
University of Applied Sciences, Hagenberg — *Hagenberg, Austria*

RESEARCH EXPERIENCE

Graduate Student Researcher — **2014–2018**
University of California, Irvine — *Irvine, California*

Intern — **Summer 2017**
Immunant, Inc. — *Irvine, California*

Research Intern — **Summer 2016**
Microsoft Research — *Cambridge, England*

TEACHING EXPERIENCE

Teaching Assistant — **Spring 2015**
ICS 46: Data Structure Implementation and Analysis (C++)
University of California, Irvine — *Irvine, California*

Teaching Assistant — **Winter 2014**
ICS 45: Programming in Java as a Second Language
University of California, Irvine — *Irvine, California*

REFEREED PUBLICATIONS

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

PartiSan: Fast and Flexible Sanitization via Run-time Partitioning **RAID 2018**
Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz
International Symposium on Research in Attacks, Intrusions and Defenses

Bytecode Corruption Attacks Are Real — And How To Defend Against Them **DIMVA 2018**
Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz
Conference on Detection of Intrusions and Malware & Vulnerability Assessment

Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory **USENIX Sec 2017**
Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa
USENIX Security Symposium

Subversive-C: Abusing and Protecting Dynamic Message Dispatch **USENIX ATC 2016**
Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz
USENIX Annual Technical Conference

SOFTWARE

TypePipe `https://github.com/re-motion/TypePipe`
A Code Generation Pipeline for .NET (Master's thesis)

ABSTRACT OF THE DISSERTATION

Finding and Mitigating Memory Corruption Errors in Systems Software

By

Julian Lettner

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Michael Franz, Chair

Obtaining secure systems software is notoriously hard. One reason is the continuing use of unsafe languages due to their efficiency, direct control over hardware resources, and developer familiarity. However, this fine-grained control comes with opportunities for mistakes and therefore invites bugs such as memory corruption errors. In the absence of an adequate defense, these bugs can be readily exploited by attackers. This fact—combined with the inevitability of exploitable bugs due to the use of unsafe languages—has spurred a large body of research with code-reuse attacks and defenses of those attacks being the most prominent line of work.

In this dissertation, we apply the principles of code-reuse (which usually targets static or jitted code) in a dynamic context, sidestepping all existing defenses. Concretely, we demonstrate state-of-the-art, whole-function code reuse by abusing the dynamic dispatch mechanism found in languages such as Objective-C. We also devise a defense scheme and apply it to the Objective-C runtime. The result is a low-overhead, drop-in replacement for the Objective-C runtime that prevents our as well as other metadata-corruption attacks.

The assumed inevitability of exploitable bugs and stringent performance constraints of systems software have steered much of the previous research on systems security towards the mitigation of the exploitation phase. Sanitization and fuzzing are industry trends that in-

stead try to weed out the bugs themselves, i.e., they tackle the cause instead of trying to mitigate the consequences. In this dissertation, we present a novel technique to increase the efficiency of sanitizers and extend their applicability via run-time partitioning. We significantly reduce the overhead of two popular compiler-based sanitizers extending their usage scenarios and increasing fuzzing throughput. Together with other recent work, our research challenges the assumption that bugs in systems software are inevitable.

Chapter 1

Introduction

1.1 Overview

Today's systems software is implemented in unsafe languages such as C and C++. The reasons for the continued use of C/C++ are multifaceted: efficiency, direct control over hardware resources, and developer familiarity. However, the fine-grained control afforded by unsafe languages comes with the burden of fine-grained responsibility. Developers have to manage memory manually and accessing invalid memory is undefined behavior. The resulting memory corruption errors are notoriously hard to track down and often lay dormant for a long time before being fixed. With plenty of opportunity for mistakes, writing bug-free systems software is therefore an exceptionally hard problem.

Many of these memory corruption errors constitute *security vulnerabilities*, that is, bugs which are exploitable by attackers. Over the years, attackers have found creative and increasingly sophisticated ways to use different classes of bugs to their advantage. This means that a previously benign bug can turn into a security vulnerability, when a new exploitation

technique is discovered. The end goal[1] of an attacker usually is *remote code execution*, i.e., the ability to execute arbitrary code on the target's machine; or the leakage of sensitive data like encryption keys or account credentials.

A typical example for an attack target is a host providing a network service. Let us assume that the service implementation—or any of its dependencies—contains an unpatched memory corruption vulnerability. The attacker might be able to send a maliciously crafted request that triggers the vulnerability, allowing the attacker to inject the exploit payload. If the execution of the payload succeeds, the attacker gains the ability to execute arbitrary code on the target host with the privileges of the service process. Note that the service itself is benign and its authors have good intentions. The attacker just happens to be able to leverage a vulnerability in its implementation to his advantage.

Another popular target for exploitation is web browsers. In this context, attackers are often able to trick unsuspecting users into downloading and executing a malicious script. To protect their users, web browsers execute untrusted scripts inside a *sandbox*. A sandbox is a restricted execution environment that limits access to certain functionality and ideally prevents a malicious script from doing real harm. However, if the sandbox or browser implementation itself contains a vulnerability, an attacker might be able to first break out of the sandbox and then continue with his attack unhindered.

The abundance of exploitable bugs in systems software in combination with millions of targets running this vulnerable software has fueled a very active feedback loop between offensive and defensive techniques. This research area is loosely termed *systems security*. From this body of research, a certain class of attacks—so-called *code-reuse attacks*—have been shown to be extremely potent. This has motivated a plethora of defenses with varying levels of protection and resource requirements.

[1]Here we mean the end goal of an attacker on the technical level. The real-world end goal might be monetary gain, exfiltration of trade secrets, etc.

In Chapter 3, we demonstrate a new offensive technique targeting Objective-C programs running on Apple's OS X and iOS platforms. It leverages whole-function code-reuse to sidestep all existing defenses. We also retrofit the Objective-C runtime with an effective defense against our attack. Our defense is able to protect complex, real-world software such as iTunes without recompilation and our performance experiments show that the overhead of our defense is low in practice.

Defenses are imperfect for a wide range of reasons. Even a bulletproof defense will not garner widespread adoption if it falls short on a single practical issue such as performance overhead, compatibility with unprotected code, or developer productivity. In academia, researchers therefore often use the word *mitigation* to reflect this sentiment. In this dissertation, we use the terms defense and mitigation interchangeably. Another witness of imperfect defenses is the concept of *defense in depth*, that is, the practice of layering defensive measures.

Besides practical issues, the primary reason for most defenses' limited shelf life is that they attempt to mitigate the consequences, i.e., thwart attacks, but do not eliminate the underlying vulnerability. In the context of our web server example from above, this means foiling the successful execution of the payload instead of preventing the attacker from injecting the payload in the first place. In many recent works, the focus has therefore shifted to finding security-critical bugs and thereby eliminating the underlying vulnerabilities. And in the broader context of systems security, other lines of research like language design, software engineering, and formal verification also aim to help developers construct bug-free programs. In this dissertation, we focus on an emerging industry trend in this area: the extensive adoption of sanitization and fuzzing as part of the development and testing workflow.

Sanitizers are dynamic bug-finding tools that help developers find bugs that elude static analysis. They are especially useful in the context of languages with undefined behavior such as C/C++ where a security-critical bug may lie dormant for a long time. Unlike static analysis, however, sanitizers can only detect errors on code paths that are actually executed.

Therefore, they are often combined with a fuzzer to drive execution. Currently, sanitizers are mostly used during development and testing. This is in large part due to the high memory and processing requirements of sanitizers. However, there is a desire to sanitize on end user systems, as indicated by recent efforts of major browser vendors [41, 42, 48].

In Chapter 4, we present PartiSan, a run-time partitioning technique that speeds up sanitizers and allows them to be used in a more flexible manner. Our core idea is to partition the execution into sanitized slices that incur a run-time overhead, and "unsanitized" slices running at full speed. With PartiSan, sanitization is no longer an all-or-nothing proposition. A single build can be distributed to every user regardless of their willingness to enable sanitization and the capabilities of their host system. PartiSan enables application developers to define their own sanitization policies. Such policies can automatically adjust the amount of sanitization to fit within a performance budget or disable sanitization if the host lacks sufficient resources. The flexibility afforded by run-time partitioning also means that we can alternate between different types of sanitizers dynamically; today, developers have to pick a single type of sanitizer ahead of time. Finally, we show that run-time partitioning can speed up fuzzing by running the sanitized partition only when the fuzzer discovers an input that causes a crash or uncovers new execution paths.

1.2 Contributions

This dissertation makes the following contributions, organized by chapter:

Chapter 3

- **Novel Offensive Technique** We present Subversive-C, a new offensive technique that reuses entire Objective-C methods by carefully arranging the metadata used to

dispatch messages in the Objective-C runtime. The dynamic nature of Objective-C coupled with whole-function reuse renders existing integrity and randomization-based defenses ineffective against Subversive-C exploits.

- **Hardened Objective-C Runtime** Because existing defenses cannot protect against Subversive-C with low overheads, we developed a new defensive technique to prevent adversaries from manipulating and corrupting metadata used by the Objective-C runtime. Specifically, we retrofit the Objective-C runtime with integrity checks in the lookup processes that handle Objective-C message dispatch. Our hardened runtime is fully compatible with the runtime shipped with OS X and can protect complex, real-world applications such as iTunes.

- **Realistic and Extensive Evaluation** We demonstrate a fully-fledged Subversive-C attack targeting the AppKit library. We also provide a careful and detailed evaluation of our hardened Objective-C runtime. We report an average overhead of 76.55 % on micro benchmarks designed to stress Objective-C message dispatch as well as a 1.54 % aggregate overhead for complex, real-world applications.

Chapter 4

- **Run-time Partitioning** We describe PartiSan, a framework to partition program execution into sanitized/unsanitized fragments at run time. Unlike previous approaches, the partitioning is not static but happens dynamically according to a policy-driven, run-time partitioning mechanism which selects the function variant to execute with low overhead. This lets developers release sanitizer-enabled builds to end users and thereby cover more execution paths.

- **Fast and Flexible Sanitization** We present a fully-fledged prototype implementation of our ideas and explore three concrete run-time partitioning policies. We combined

PartiSan with two popular sanitizers, ASan and UBSan, which are used by thousands of developers every day. Finally, we measured the performance overhead of our prototype on the SPEC CPU 2006 benchmark suite with our expected-cost partitioning policy.

- **Security and Performance Evaluation** We present a thorough evaluation showing that our approach still detects the majority of vulnerabilities at greatly reduced performance overheads. For the popular ASan and UBSan sanitizers, PartiSan reduces overheads by 68 % and 76 % respectively.

- **Improved Fuzzing Efficiency** We demonstrate an important use case of PartiSan: improving fuzzing efficiency. We combined PartiSan with a popular fuzzer and measured consistently increased fuzzing throughput.

Chapter 2

Background

2.1 Memory Corruption

As we motivated in the introduction, systems software is prone to memory corruption errors due to the use of unsafe languages. The literature classifies these errors either as spatial or temporal memory safety violations [8, 67, 68]. Examples of spatial errors include *out-of-bound* array accesses and the dereference of uninitialized or invalid pointers. More formally, a spatial memory safety violation is caused by dereferencing a pointer that points outside the bounds of its referent. In contrast, a temporal violation is caused by dereferencing a pointer whose referent has been deallocated. The most common class of temporal memory errors is accesses to `free`'d memory, e.g., *use-after-free* and *double-free* bugs. Another example is references to automatically allocated memory with an expired lifetime, such as references to stack variables that escape the containing function. In the absence of any spatial and temporal memory errors a program is said to be memory safe.

Although many measures have been taken to detect memory safety violations [3, 85], they are still discovered in modern software [93, 98]. Memory corruption attacks exploit these

memory-related bugs to subvert the control flow of an application, and trigger malicious actions thereafter. As a consequence, countermeasures have been proposed to prevent the malicious computation after the initial overwrite. However, as attacks become more sophisticated to overcome deployed defenses, the definition of what constitutes a security vulnerability changes with them. For example, the use of uninitialized memory—previously thought of mostly benign—has been identified as a major attack vector against the Linux kernel [61, 64]. In the following sections, we give an overview of the evolution of attack techniques (and the co-evolution of their respective defenses) from simple code injection, to return-oriented programming, to state-of-the-art code-reuse attacks.

2.2 Code Injection

While all memory corruption errors constitute program reliability and safety issues, not all of them are necessarily security vulnerabilities. For a memory bug to be exploitable, an attacker generally needs to be able to perform the two following steps:

1. Trigger and use the memory corruption bug to inject an attack payload.

2. Hijack control flow to divert execution to the attack payload.

The classic stack smashing attack accomplishes both of these steps via a single sequential buffer overflow on the stack [57]. For this attack, the payload used to overflow the vulnerable buffer directly encodes the attack code. In other words, this attack works by injecting executable bytes into the memory space of the target process, hence the name *code injection*. The same buffer overflow is used to overwrite the return address of a currently executing function, which is located in the function's stack frame on the call stack. The attacker chooses the new value so that it points back to the payload in the overflowed buffer. Eventually, when

the return statement of the associated function executes, the corrupted value is interpreted as the return address redirecting execution to the injected payload.

Today, traditional code injection attacks are prevented by widely deployed defenses such as data execution prevention (DEP) [4] and write-xor-execute (W⊕X) [74]. Note that the described attack injects code on the function call stack. In general, memory sections for the stack, heap, and other data do not contain legitimate code. DEP therefore marks data pages non-executable, which prevents the execution of the injected code. The W⊕X policy enforces a combination of non-executable data and code integrity; pages can be mapped as either writable or executable, but not both. Today, all modern CPUs support non-executable page permissions, so defenses against code injection are fast and effective.

2.3 Code Reuse

Since DEP prevents the injection of new code, attackers have shifted to reusing existing code. The systems security literature calls the repurposing of legitimate program code for malicious purposes *code reuse* [75].

The most popular instantiation of the concept of code reuse is return-oriented programming (ROP) [82]. For ROP, the attacker combines short code sequences called *gadgets* from the executable or linked libraries to craft an exploit. To allow gadgets to execute one after another, the leveraged code sequences need to terminate with a return instruction, hence the name. These return instructions fetch the next pointer from an attacker-controlled stack to redirect execution to the subsequent gadget in the code-reuse payload. Typically, the attacker performs static analysis on the executable and linked libraries to identify useful code sequences and gadgets. Thereafter, the attacker generates a code-reuse payload that only consists of code pointers which reference the selected gadgets and some data values

processed during gadget execution.

In addition to ROP, there are various other known code-reuse techniques. The main difference between these techniques is the granularity at which legitimate code is reused. One of the earliest published attacks, called return-into-`libc` (RILC), uses a return instruction to call into dynamically linked library functions [70, 95]. Another attack is similar to ROP, but leverages gadgets ending in indirect jump/call instructions instead of returns [18]. Finally, our attack presented in Chapter 3 is inspired by counterfeit object-oriented programming (COOP) [76]. This kind of attack chains together dynamically-bound functions in C++, that is, virtual functions, to construct exploit semantics. See Section 3.2.3 for a detailed explanation of COOP.

Defenses against code-reuse attacks usually fall into one of two categories: *randomization*-based or *integrity*-based mitigations. The goal of randomization-based defenses is to present attackers with an unpredictable attack surface, making their a priori knowledge about a vulnerable binary obsolete and thus preventing them from crafting a successful attack payload. In a code-reuse attack the payload necessarily consists of (direct or indirect) references to existing code, e.g., gadget addresses. The location of these gadgets—or more broadly, the target process's memory layout—is therefore an important prerequisite of a code-reuse attack. Address space layout randomization (ASLR) makes it harder for attackers to predict the location of gadgets by randomizing the base address of the executable, linked libraries and the positions of the stack and heap [71]. Analogously to DEP, some variant of ASLR is enabled by default on every modern operating system.

Unfortunately, the effectiveness of ASLR is undermined by *information leakage* and *memory disclosure attacks* [81, 83, 87]. The disclosure of a single code pointer referencing a known location (such as a library function) enables the attacker to derandomize the containing library due to ASLR's limited randomization granularity. This means that the attacker can dynamically compute all gadget addresses inside the same library, because ASLR only

randomizes base addresses. Other, more fine-grained randomization schemes are described in the literature [43, 53], but rarely deployed due to practical reasons.

Integrity-based defenses attempt to enforce high-level program semantics on the architectural or machine level which typically offers a much higher degree of freedom. Attackers often exploit this semantic gap between the higher and lower levels of computing. For example, ROP adds control-flow transfers to chain the execution of return-terminated gadgets. These additional transfers are absent from the control-flow graph (CFG) of the original program and unintended by the programmer. The goal of control-flow integrity (CFI) is to restrict the control flow of an application to its predetermined CFG to prevent unintended transfers [1]. Most CFI implementations use static analysis to determine a program's CFG. For each indirect call site in the program source, this analysis computes a set of possible targets at run time. Subsequently, runtime checks are added to all indirect call sites to ensure that the called function is contained in the set of valid targets. This protects the integrity of call sites and jumps, also called *forward edges*. Function returns, or *backward edges*, are usually protected by a *shadow stack*, which ensures the integrity of important control data such as the return address [23].

Since typical CFI implementations simply abort the program if they detect an unintended control-flow transfer and static analysis is an undecidable problem in the general case [52], the analysis has to be conservative. Therefore, the obtained CFG is usually an overapproximation of the real, underlying CFG. This leaves some leeway between intended and unintended control-flow transfers which has been shown to provide a sufficient degree of freedom to mount meaningful attacks within the bounds of coarse-grained CFI implementations [25, 76]. Similar to randomization-based defenses, this back-and-forth between attackers and defenders has sparked much research exploring the security and performance trade-offs for a range of coarse and fine-grained CFI policies [15, 16, 33, 66, 97]. Today, basic CFI policies are supported by most mature C/C++ compilers such as gcc, Clang, and MSVC [63, 94].

2.4 Fuzzing

In recent years, the concept of fuzz testing or *fuzzing* has gained widespread industry adoption [78]. It has become an integral part of a disciplined software engineering process and is especially useful to discover vulnerabilities such as memory corruption errors. When first introduced in 1990, the term fuzzing was used to describe a testing technique in which the target program was executed with random input data [65]. Today, a plethora of fuzzing techniques exist to guide the test case generation process. Efficient, mutation-based fuzzers rely on fast execution and high test throughput since they will often choose *uninteresting* program inputs [60, 92, 105]. More sophisticated, guided fuzzers leverage advanced techniques such as symbolic execution, but suffer from state explosion and other practical scalability issues [36, 91].

Probably the most popular security-oriented fuzzer is American fuzzy lop (AFL) [105]. With AFL, the program under test is compiled with coverage instrumentation to provide feedback to the fuzzer. Honggfuzz is another security-oriented fuzzer that supports additional forms of coverage feedback such as hardware-based counters or execution traces collected with Intel PT [92]. To help with vulnerability discovery, the program under test usually has further instrumentations applied to it. Prominent examples of instrumentation that help with detecting memory corruption errors are sanitizers such as AddressSanitizer [79], UndefinedBehaviourSanitizer [59], and MemorySanitizer [90]. In Section 4.7 we combine our work on sanitizers with libFuzzer, which is a coverage-guided, evolutionary fuzzer [60]. Figure 2.1 illustrates the most important steps in a coverage-guided, evolutionary fuzzing loop:

1. The developer provides a seed corpus, usually consisting of a few examples for valid program input. This increases the efficiency of fuzzing, although most fuzzers also support starting from an empty corpus.

2. The fuzzer picks a random input from the corpus. Inputs may optionally have weights

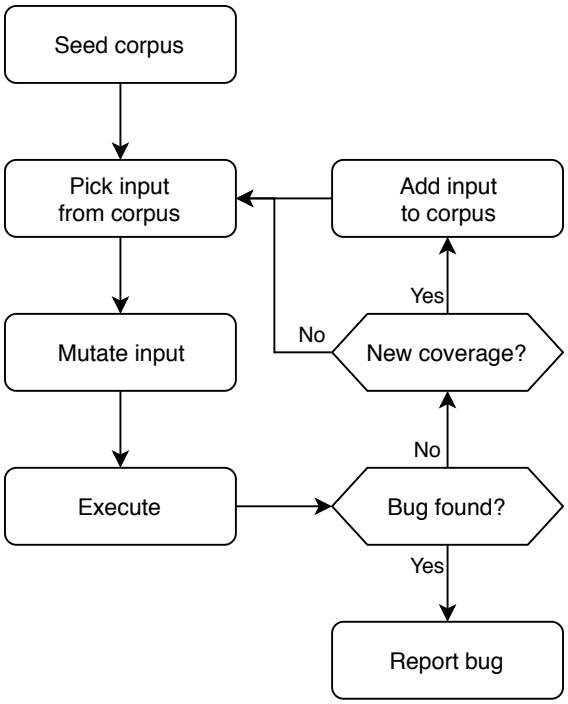

Figure 2.1: Main loop for coverage-guided, evolutionary fuzzing

associated with them, so that an input which produced interesting offspring has a higher chance of being selected again.

3. The selected input undergoes a sequence of mutations to derive a new input. This resembles the evolutionary step in *genetic programming* while the inputs in the corpus serve as the population [49].

4. The fuzzer executes the program under test with the newly derived input. To collect coverage feedback during runs, the program itself is instrumented for that purpose. Usually, the program will also have one or more sanitizers applied to it to increase the chance of finding a bug.

5. If a bug is found during execution, the fuzzer reports the crash and corresponding input. The developer can now use the offending input to reproduce the crash and fix the bug. Afterwards, the fuzzing process is started anew.

6. If no bug is found, the fuzzer checks if the current input exercised any new code paths

int the program. If so, the input is deemed *interesting* and added back into the corpus, or discarded otherwise. The fuzzing loop now continues with step 2.

Chapter 3

Abusing and Protecting Dynamic Message Dispatch

3.1 Motivation

Before the introduction of Swift, the primary programming environment on Apple's OS X and iOS platforms used a language called Objective-C, which extends the C language with object-oriented constructs. Even today, many of the main application programs on Apple's platforms, such as Safari, iTunes, etc. are built predominantly using Objective-C, which differs from C++ in the way that dynamic dispatch of function calls is implemented. In spite of its importance to commercial software platforms, it has attracted little scrutiny from systems security researchers.

Remote exploitation of memory corruption vulnerabilities remains the greatest threat facing systems software written in C, C++, and Objective-C. Thanks to the now widespread deployment of mitigations such as DEP, stack canaries, and ASLR some exploitation techniques are no longer viable (code injection [57]) while techniques that remain viable require

more involved vulnerabilities such as use-after-free errors or heap-based buffer overflows.

The latest code-reuse mitigation being deployed—CFI—makes traditional ROP [75] attacks harder to construct. CFI computes an approximation of an application's CFG and verifies that all indirect branches follow valid CFG edges at run time [2]. In contrast to randomization-based defenses [53], CFI is secretless and cannot be bypassed via information leakage. Like other mitigations, CFI must trade off security (precision) for performance. Coarse-grained CFI policies [106, 107] leave a small fraction of code locations available for reuse by adversaries—enough to mount ROP attacks [25, 37, 77]. The deficiencies of coarse-grained CFI renewed interest in more precise policies. Devising such CFI policies typically requires source code access, because structural information required to compute a complete and precise CFG is lost during compilation.

The recent COOP [76] code-reuse technique exploits the imprecision of non-C++ aware CFI implementations on Windows and Linux. Specifically, the attacker manipulates the *virtual method tables (vtables)* of C++ objects in memory such that a sequence of attacker-chosen regular virtual methods is executed via likewise regular virtual method call sites. Unlike ROP, COOP does not violate the integrity of return addresses or produce corrupted call stacks and therefore remains undetected by generic CFI policies [27, 34]. Moreover, the high-level structure of C++ code (e.g., class hierarchy and dynamic object types) cannot be fully recovered without source code, so malicious COOP control flows are difficult to distinguish from benign ones even for *C++-aware* CFI policies computed by binary analysis. In terms of expressiveness and flexibility, COOP is comparable to ROP in C++ environments [22, 76]. Still, it remains unclear whether COOP is limited to C++ code on Windows and Linux or whether it is a generic threat on par with ROP.

Our work shows that programs written in Objective-C suffer from a systematic vulnerability that enables COOP-style exploits against Objective-C on OS X and iOS. Like C++, Objective-C extends the C programming language with object-oriented constructs. Al-

though both languages add *dynamic dispatch* of function calls to C, the implementation of this feature differs greatly between C++ and Objective-C. Whereas C++ fixes the vtable for each class at compile time, Objective-C enables full *late binding* by (re)mapping literal method names to actual functions dynamically at run time. We dub our new class of attacks Subversive-C and demonstrate its viability against applications using AppKit, a commonly used framework on Mac OS X, by constructing a proof-of-concept exploit.

We also show how Subversive-C exploits can be mitigated. Our mitigation strategy can be retrofitted onto existing systems without requiring recompilation of the programs being protected and has very little overhead. In particular, we designed, implemented, and evaluated a defense that protects the integrity of core Objective-C runtime data structures.

An important insight is that in many cases, an attacker can use COOP, Subversive-C, or a combination of both, because non-trivial OS X and iOS applications like Safari or MS Office typically contain both Objective-C and C++ (standard libraries or ported code from other platforms) components. In fact, it is even valid (and common) to tightly interweave Objective-C and C++ semantics. Such "Objective-C++" code is accepted by the GCC and Clang compilers. Hence, effective code-reuse defenses for OS X and iOS need not only to consider high-level semantics of Objective-C, but also those of C++.

3.2 Background

In the following, we provide a brief overview of the technical concepts we use in the rest of this work. We discuss dynamic message dispatch in Objective-C and present an overview of resarch on code-reuse attacks with a specific focus on COOP.

3.2.1 Dynamic Message Dispatch

Objective-C is an object-oriented programming language that extends the C language with dynamically dispatched Smalltalk-style messaging. Where C++ programmers invoke (virtual) methods of objects, Objective-C programmers send messages to objects. Each message has three components: *(i)* the *receiver* object; *(ii)* the *selector*, an identifier naming the method that receives the message; and *(iii)* zero or more *arguments*.

Listing 3.1 shows two examples of Objective-C message dispatch. In line 1, the program sends a zero-argument message named `foo` to the first object. In line 2, the program sends a message `bar` with two arguments.

Listing 3.1: Objective-C message dispatch examples

```
1    [obj1 foo];
2    [obj2 bar:1 arg:2];
```

Although Objective-C is a statically compiled language, the targets of message dispatches are resolved at run time. At every message dispatch location, the compiler simply emits a call to the `msgSend` function (or one of its variants) in the Objective-C runtime. The purpose of the `msgSend` function is to locate the appropriate method for a given (receiver, selector) pair and subsequently execute it.

Figure 3.1 illustrates the message dispatch algorithm as implemented in Apple's Objective-C runtime. It consists of a fast path and a slow path. The slow path retrieves the method implementation corresponding to a given selector by searching through all methods defined by the class of the receiver object and all its ancestors. The search operates on compiler-generated metadata attached to each object as shown in Figure 3.2.

The lookup algorithm starts with the class of the receiver and checks the selector against all methods defined by the receiver's class. If no method is found, the methods of the parent

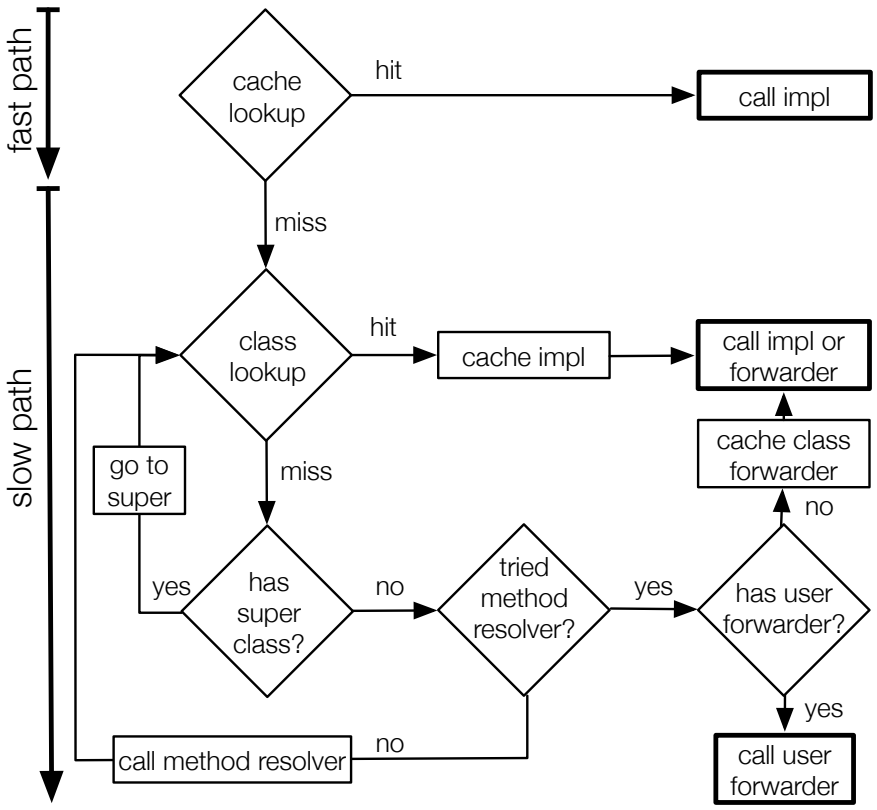

Figure 3.1: Fast and slow paths when dispatching messages

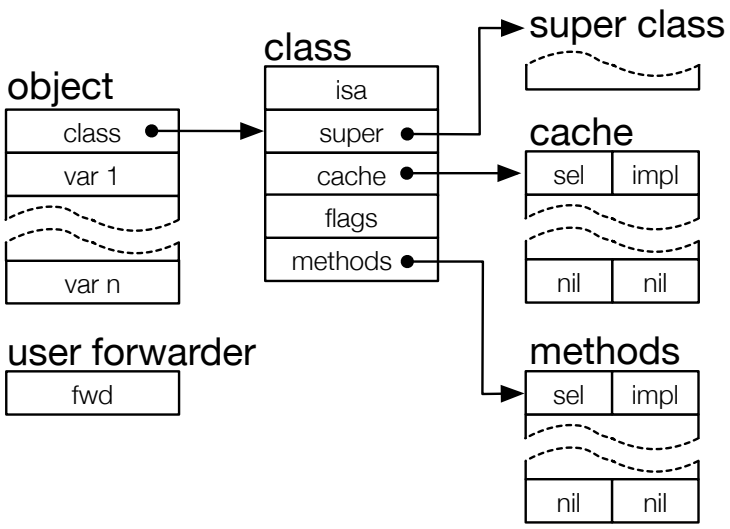

Figure 3.2: Layout of objects, classes, and lookup caches

class is searched until a method implementation is found or the root of the class hierarchy is reached. If neither the class itself nor any of its ancestors contain a method implementation, the runtime allows the class to dynamically add an implementation for the given selector. If the class provides a "method resolver" function, the runtime calls it. The resolver (depending on its implementation) may add a new method to the class. The runtime then repeats the entire lookup process, in case the method added by the resolver corresponds to the input selector.

The fast path speeds up message dispatch by storing the results of each lookup in a per-class cache, and reuses previous results if available. At the beginning of each message dispatch, the runtime queries this cache for a method pointer as shown near the top of Figure 3.1.

The cache entries are stored in memory as a linear array of (selector, method pointer) pairs. The class metadata includes a pointer to its corresponding cache. The runtime performs the lookup using a linear probing algorithm. The lookup starts from a location computed by hashing the selector itself and proceeds linearly through the array until either a match is found—a *cache hit*—or an empty entry is reached—a *cache miss*. Listing 3.2 shows the cache lookup algorithm in pseudo code format. Note that this algorithm is implemented

Listing 3.2: Cache lookup algorithm

```
id objc_msgSend(Object obj, Selector sel) {
  class = obj->class;
  cache = class->cache;
  b = cache_hash(sel) % cache->size;
  while (cache->buckets[b] != NULL) {
    if (cache->buckets[b]->sel == sel) {
      // Cache hit
      impl = cache->buckets[b]->impl;
      jmp impl;
    }
    b = (b + 1) % cache->size;
  }
CacheMiss:
  // Take the slow path
}
```

with hand-tuned assembly code in the actual Objective-C runtime.

In case neither the cache lookup nor the slow class hierarchy walk find the method for a selector, the runtime performs one last step before exiting with an error. If the class provides a "forwarder" function, the runtime calls this function with the selector, allowing classes to dynamically respond to new messages at run time or forward messages to other objects. Additionally, the Objective-C runtime allows the application to install a "user forwarder" that overrides all per-class forwarders. If this handler is installed, the runtime calls it at the end of the method lookup process. The handler pointer is stored in a writable global variable, which can be manipulated by adversaries.

3.2.2 Exploitation and Code Reuse

C/C++/Objective-C eschew memory and type safety features of modern languages and require manual memory allocation and deallocation. This leads to a steady stream of memory

management errors[1]. Attackers exploit the presence of these errors to craft malicious inputs that hijack the control flow of the application. The classic stack smashing attack injects code and redirects execution to it by overwriting the return address stored past the end of an overflowed buffer [57]. Thanks to modern mitigations such as DEP, which disallows memory regions that are both writable and executable, code injection is all but obsolete. Therefore, modern exploits reuse legitimate code to bypass DEP. There are many known variants of code-reuse attacks. The main differences are the granularity at which legitimate code is reused. ROP reuses short instruction sequences ending in returns called gadgets [75, 82]. Other variants reuse whole functions, including RILC, COOP, and our novel Subversive-C technique. Another key difference is the dispatching mechanism used to transfer control between the code snippets being reused. ROP and RILC use return instructions or indirect jump/call instructions [18]. COOP-style attacks use special "main-loop gadgets" to iteratively or recursively dispatch a sequence of method calls controlled by a malicious payload.

An important prerequisite of a code-reuse attack is knowledge of the target's memory layout, because the payload in a code-reuse attack necessarily (directly or indirectly) references existing code locations. Thus, ASLR complicates code-reuse attacks because it randomizes the base address of linked libraries at load time. However, this is only a small hurdle for a practical code-reuse attack since information leakage or memory disclosure attacks often enable attackers to undermine ASLR [81, 83, 84, 87]. Once a run-time address of a certain library function is known, the attacker can dynamically calculate all the gadget addresses inside the same library, because ASLR only randomizes the base address. Furthermore, the so-called just-in-time code-reuse attacks (JIT-ROP) can dynamically search for gadgets even in fine-grained randomized code without any static analysis [87]. As a result, code-reuse attacks have evolved to the state-of-the-art exploitation technique frequently leveraged in real-world attacks against software.

[1]For new code, disciplined use of modern coding techniques like smart pointers for C++ and automatic reference counting (ARC) for Objective-C alleviate this problem.

3.2.3 Counterfeit Object-Oriented Programming

COOP is a code-reuse technique targeting C++ software [76]. In COOP, a sequence of attacker-chosen C++ virtual methods (also called *vfgadgets*) is executed on attacker-injected objects (also called *counterfeit objects*). Each vfgadget in such a sequence fulfills a specific task, such as reading a value into a register and may have certain side-effects. Executed one after another, the vfgadgets implement the malicious functionality desired by the attacker, e.g., the execution of a shell command. Put simply, (short) virtual methods are to COOP attacks what gadgets are to ROP attacks. Whereas a ROP attack is initiated by injecting a "fake stack" (containing fake return addresses) into the target address space, a COOP attack injects a collection of "counterfeit objects", typically using a single attacker-controlled write. Each counterfeit object corresponds to exactly one vfgadget and carries a corresponding pointer to a vtable[2].

In ROP and related techniques, data primarily flows through registers and the stack from one gadget to another. In contrast, data may flow in three different ways between COOP vfgadgets: *(i)* through method arguments, *(ii)* through global variables, and *(iii)* through member fields of *overlapping* counterfeit objects. The latter is a pattern specific to COOP, which can greatly facilitate the creation of meaningful vfgadget chains. For example, vfgadget *1* may read a value from memory and store it in the field x of counterfeit object A and vfgadget *2* may increment field y of counterfeit object B. By making objects A and B overlap such that $A.x$ and $B.y$ map to the same address, the methods *1* and *2* can be used in conjunction to read and increment a value.

Different techniques for chaining the execution of vfgadgets in a COOP attack have been described in previous work [22, 76]. Using one of these techniques, the attacker initially

[2]In C++, every object of a class with virtual methods carries a pointer to the class's fixed vtable. Whenever a virtual method is to be executed on a C++ object at run time, this pointer is dereferenced and the respective method's address is fetched from the table.

corrupts a C++ object used by the target application such that a subsequent virtual method call is maliciously diverted to a central dispatcher vfgadget. (In a ROP attack, the control flow would instead be diverted to the first gadget, which usually pivots the stack pointer.) In the simplest case, a so called *main loop* (ML-G) vfgadget is used. Briefly, an ML-G is a virtual method that invokes virtual methods on a collection of C++ objects. By making an ML-G operate on a collection of counterfeit objects, the chained execution of vfgadgets becomes possible. An example ML-G is discussed in Section 3.4.2.

3.3 Threat Model and Assumptions

The assumptions underpinning this research are:

- **Data** The attacker can arbitrarily read and write data pages as allowed by the page permissions. Specifically, the internal data structures of the Objective-C runtime can be read, written, and corrupted. However, we assume that ASLR is in place to randomize the locations of program and runtime data structures.
- **Code** We assume that DEP prevents code injection by disallowing the execution of writable pages.
- **Runtime** We assume that the runtime is protected using fine-grained code randomization [53], as well as an implementation of execute-only memory (XoM), such as XnR [9], Readactor [21], or HideM [35], that prevents attackers from using information leaks to retrieve the code of the runtime. We also rely on these defenses to securely store secret keys inside XoM, thereby preventing the attacker from reading the keys (see Section 3.5.4).
- **Control flow** Since Objective-C is a superset of C, we assume the C parts of the application and runtime are protected using appropriate mitigations (CFI, randomization,

or equivalent defenses). More specifically, we assume that the attacker has no other way of hijacking control flow. Defenses such as Mobile CFI (MoCFI) [24] can be used to protect Objective-C code from control-flow hijacking.

Note that these assumptions are realistic and match the capabilities of a real-world attacker. They also match the adversarial models used in closely related work [21, 22].

3.4 Subversive-C

In this section, we demonstrate that the principles of the COOP attack are not only applicable to C++ but also to Objective-C. Conceptually, a Subversive-C attack proceeds analogously to a COOP attack: the attacker diverts an application's control flow such that a sequence of attacker-chosen Objective-C methods (vfgadgets) is executed on injected counterfeit objects. The first method executed in such a sequence is necessarily a dispatcher vfgadget, e.g., a *main loop vfgadget* (ML-G) as described in Section 3.2.3. COOP and Subversive-C are closely related in the way they rely on counterfeit objects and vfgadgets. However, as they target different programming languages, COOP and Subversive-C counterfeit objects differ in their layouts. For COOP it is sufficient to create objects that reference a vtable, whereas the Objective-C runtime features a more involved class layout. Therefore, an attacker must forge multiple data structures to launch a Subversive-C attack. The exact procedure is described next in Section 3.4.1. Section 3.4.2 then presents a concrete Subversive-C attack against applications that use the AppKit library.

For brevity, we limit the discussions in this section to Apple's OS X operating system and the x86-64 architecture. However, all techniques and concepts extend to Objective-C code running on iOS and ARM.

3.4.1 Exploiting the Objective-C Message Dispatch Mechanism

The Objective-C runtime implements two different ways (*slow* and *fast*, see Section 3.2.1) to resolve a class-selector pair to a function address. We now describe how the attacker can exploit the Objective-C runtime's slow path and fast path lookup mechanisms in order to control the methods invoked on counterfeit objects in a Subversive-C attack. These techniques are specific to Subversive-C and are the key differentiators with respect to COOP.

Slow Path As described in Section 3.2.1, when a cache lookup for a selector fails, the `msgSend` function does a slow search through all methods available for the receiver object. The corresponding data structures are partly stored in read-only memory and cannot be modified by the attacker at run time. Hence, in order to freely choose the vfgadgets executed in a Subversive-C attack, the attacker needs to inject new fake data structures alongside each counterfeit object. Concretely, each counterfeit object needs to reference its own fake *class struct*[3] which in turn references its own fake *method list* (see Figure 3.2).

Each entry in a class's method list links a function pointer to a selector. It is thus sufficient to inject fake method lists with a single entry. This entry must link the fixed selector used in the dispatcher gadget to the vfgadget that is to be executed on the corresponding counterfeit object. In turn, the injected fake class struct must be shaped in such a way that `msgSend` actually takes the slow path and evaluates the given method list as desired. A straightforward way to ensure this is to null-out the cache-related fields in the class struct (i.e., invalidate the cache) and to mark the class as *initialized* by setting the corresponding bit in the `flags` field (not shown in Figure 3.2).

Instead of creating valid class structs from scratch, for increased stealthiness and simplicity,

[3]In practice, the class struct is oftentimes split into a read/write and a read-only part by the compiler. For brevity, we do not make a distinction between the two here and consider them as one coherent data structure.

an existing class struct that is compatible with the given dispatcher can be copied and modified as needed.

Fast Path Instead of invalidating the cache of counterfeit objects, the attacker can also opt to exploit the fast path look-up by injecting fake class structs with *valid* cache entries linking the dispatcher's selector to vfgadgets. Doing so is simple, as the caching mechanism does not use a secure hashing function and, in any case, its parameters can also be directly rewritten by the attacker. Hence, the attacker can arbitrarily precompute valid cache entries offline and incorporate them into fake class structs.

Forwarders In addition to forging method lists and caches, a third option for the attacker to execute arbitrary methods from a dispatcher is to abuse *forwarders*, which are introduced in Section 3.2.1: existing forwarders structs (see Figure 3.2) could be manipulated or fake ones could be injected such that vfgadgets are executed instead of actual forwarder handlers. In this approach, the attacker needs to make sure that both the slow and the fast path fail for all counterfeit objects for the given dispatcher—e.g., by injecting fake class structs with an invalid cache and an empty method list.

3.4.2 Proof-of-Concept Exploit

To demonstrate the general applicability of our technique, we constructed a Subversive-C attack for the x86-64 version of the AppKit library. AppKit is part of the Cocoa framework which encompasses Foundation, AppKit and Core Data. These libraries provide the starting point for Objective-C applications and AppKit in particular is used to create graphical user interfaces. As such it is included in most graphical Objective-C programs, including iTunes, Safari, Pages, Keynote, and many other widely used applications from Apple and third parties. The Objective-C methods used in the attack are given in Table 3.1. We extended

Table 3.1: Our Subversive-C chain in the standard OS X AppKit library (x86-64) calculates the address of `system()` in `libsystem` and invokes `system("/bin/sh")`. Gadget type names are according to previous work [76].

#	Method name (AppKit)	Type	Description
1	`[NSTextReplacementNode dealloc]()`	ML-G	Main loop
2	`[NSUndoTextOperation affectedRange]()`	LOAD-R64-G	Load `rdx` from instance var.
3	`[NSPersistentUIRecord setEncryptionKey:](uint8_t[16])`	R-G	Load `rdx` from address `rdx+8`
4	`[NSPanelController stringValue]()`	LOAD-R64-G	Load `rcx` from instance var.
5	`[NSMatrix cellAtRow:column:](int64_t, int64_t)`	ARITH-G	`rdx = rdx · [self+0xf8] + rcx`
6	`[NSScrollingScoreKeeper setHoldCount:](int64_t)`	W-G	Write `rdx` to instance var.
7	`[NSCustomReleaseData dealloc]()`	INV-G	Invoke instance var. as function pointer

the framework that Schuster et al. [76] used to create the COOP chains to account for the differences between C++ and Objective-C. The framework uses the SMT solver Z3 [29] to construct a buffer with the constraints defined by the layout of the objects and their required relative offsets to each other. (Recall that typically at least some counterfeit objects overlap.)

For our proof-of-concept exploit, we require a program that contains a memory corruption vulnerability allowing an attacker to place data in the target process as well as overwrite a pointer to an Objective-C instance used during execution. To reliably bypass ASLR, we further require an information leak to disclose the position of the data injected and the location of the instance pointer we override with our own counterfeit object. Since our gadgets are sourced from the AppKit library, this library must also be loaded by the target process. We simulate a suitable vulnerable application by creating an Objective-C program that requires the AppKit library and lets us inject attacker-controlled data in the address space. This data is then interpreted as an Objective-C object, more precisely as our *initial object*, which will start our chain. After this first dispatch the execution is driven entirely by our counterfeit objects.

High Level Overview For our proof of concept we opted to construct a chain that leads to the use of an *invoke gadget* to call an arbitrary function, in this case we chose `system()`. The other gadgets are used to prepare the call by calculating the function address based

on import address table (IAT) entries and arranging arguments in memory correctly. After injecting the counterfeit objects into the target process, the chain is started by dispatching a message on the *initial object*, which directs the control flow to the *main loop gadget*. This ML-G dispatches calls to all other gadgets that perform the necessary computations. The chain reads the address of `libsystem!strlen()` from the IAT and adds a precomputed offset to it. The result is then used as the target for the *invocation gadget* (INV-G in COOP parlance [76]). The argument for this call is also located in the attacker-controlled memory and is passed as well. Any precomputed data is passed via fields in the injected counterfeit objects. In Objective-C, an object's fields are also referred to as its *instance variables*.

Initial Object The initial object is the first counterfeit object and is not part of the actual chain. It is designed such that dispatching the corresponding selector on it will enter the ML-G instead of the intended function. We pass the required arguments (in this case the gadget addresses) by setting the instance variables in this object.

Main Loop At the core of our attack lies the main loop gadget. We use an array-based ML-G (row 1 in Table 3.1) which iterates over an array of objects and dispatches a constant selector on every entry. Each counterfeit object is an entry in this array. The pseudo code representation of our ML-G is shown in Listing 3.3; line 4 invokes the selector `release` on every counterfeit object in the injected array. While this particular ML-G is limited to 28 entries in a single array, inserting the ML-G itself again as the 28th entry allows the chaining of more gadgets.

Read Gadget We use two read gadgets (#2 and #4) to load `rcx` and `rdx` from instance variables. As these are *argument registers*, they are guaranteed to remain unaltered by `msgSend`. We load `rdx` with the address of the IAT entry of `strlen()` and `rcx` with the offset between `strlen()` and `system()` in `libsystem`.

Listing 3.3: ML-G in method `dealloc()` of class `NSTextReplacementNode`

```
1 children = self->children;
2 counter = 0;
3 while (children[counter] != NULL && counter < 28) {
4   [children[counter] release];
5   counter++;
6 }
```

Read Gadget with Dereference As we only assume the address of the AppKit module to be given, the address of `system()` in `libsystem` needs to be calculated dynamically. To this end, we read a pointer to `libsystem` from the IAT of the AppKit module and, in the next step, add a constant offset to it. The gadget we use (row 3) loads `rdx` with the 64-bit value pointed to by `rdx+8`. As we control the value of `rdx` with gadget #2, we can read from a chosen address here. We use this to load `rdx` with the address of `strlen()` from AppKit's IAT.

We also discovered an additional method to directly read or write arbitrary memory locations. For each instance variable of an Objective-C class, the offset from the class pointer is stored in a writable memory region. This means we can craft a counterfeit object of a specific class and overwrite the corresponding variable offsets. Using this method an instance variable can exist anywhere in the process space. This modification can be performed by write gadgets from within a Subversive-C chain.

Arithmetic Gadget At this point `rdx` and `rcx` contain attacker-controlled values and can be used to calculate the actual address of `system()`. Gadget #5 adds both registers and stores the result to `rdx`.

Store Gadget Due to the semantics of our *invocation gadget* (INV-G) (see next step) we need to store the calculated address of `system()` in a specific instance variable of counterfeit object #7. Thus, the two counterfeit objects corresponding to gadgets #6 and #7 need to

overlap: gadget #6 stores the function pointer in `rdx` in an instance variable of its counterfeit object; gadget #7 reads this pointer from an instance variable in its counterfeit object (which maps to the same address) and invokes it.

Invocation Gadget The original purpose of our INV-G (#7) is the invocation of a custom deallocator specified via an instance variable. The argument that is passed is also read from an instance variable. This means we both control the function called and its argument. Here, we use this to execute `system("/bin/sh")`, which completes our proof-of-concept exploit.

3.5 Mitigating Subversive-C

A key insight of our attack is that it targets data structures specific to the Objective-C runtime, much like COOP targets the C++ specific vtable. Therefore, we build our defense around protecting the integrity of these data structures. Unlike C++ vtables, the data structures used by `msgSend` are mutable which means COOP defenses such as vtable randomization [22] are not suitable to protect the Objective-C runtime against Subversive-C. Instead, we choose to base our defense on message authentication to detect malicious tampering.

We add a message authentication code (MAC) to every sensitive field or data structure in the runtime as shown in Figure 3.3, and use this MAC to verify the integrity of the data structures before sensitive control flow transfers, i.e., those that indirectly use the contents of the data structures. Every time the runtime changes the contents of one of its structures, it also updates the MAC. Thus, an attacker can no longer alter its data structures undetected without also updating the associated MAC. However, each MAC computation has two inputs: the message (data) to authenticate and a secret key. Without both inputs, a correct MAC cannot be computed. Knowing the secret keys would allow attackers to tamper with runtime data structures, so we store them in a key store which attackers cannot read. We

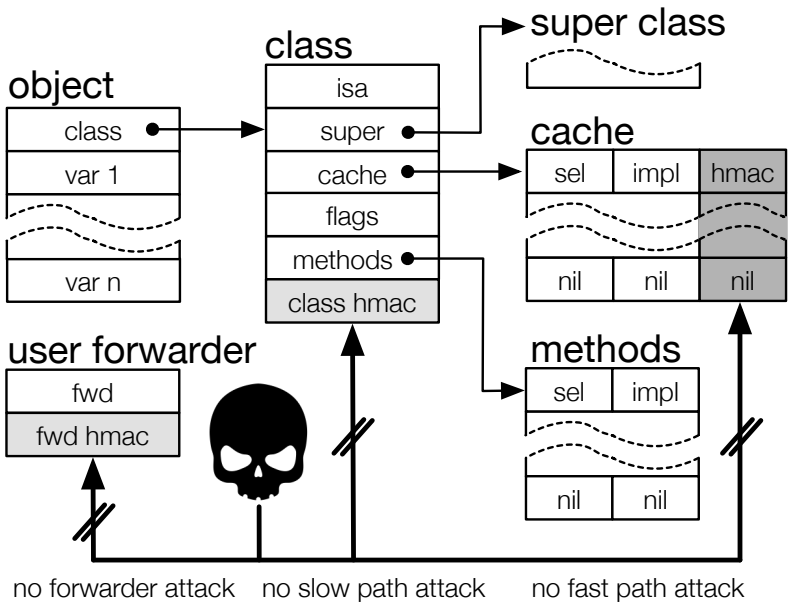

Figure 3.3: HMACs are used to ensure the integrity of class metadata and caches

describe the key store in detail in Section 3.5.4.

In the following, we first describe our different approaches to the stages of method lookup, as they have different requirements (most notably the tolerable overhead). Subsequently, we explain the implementation of our secure key store which protects keys from attackers.

3.5.1 Securing the Slow Path

To protect the slow path lookup, we repurpose an unused field in the `class` structure to store a MAC, or more precisely a hash-based MAC (HMAC), as depicted in Figure 3.3. The hash is populated during class initialization and checked before any class metadata is used for method lookup. If a discrepancy is detected, program execution is aborted with an error message. To compute the hash, we chose the HMAC-MD5 function with the following inputs:

- The **method list** consisting of flags, entry count and an array of `method` structures.

- The **superclass** field to prevent the attacker from modifying the class hierarchy.
- The **flags** field to prevent the attacker from removing the *initialized* bit. An unset *initialized* bit forces the runtime to rebuild the method list (a process which the attacker could tamper with).
- The **isa** field which points to the meta-class of the current class.
- The **address** of the `class` object to uniquely identify the class. A unique identifier is needed to distinguish between similar classes, such as siblings, preventing the attacker from copying the method lists and hash values between them (in such cases, the superclass pointer and flags match).
- A **secret key**—K_{class}—retrieved from a secure key store, which we discuss in more detail in Section 3.5.4. We use a single global K_{class} for all classes in the application.

Let X be the concatenation of all the elements in the above list except the secret key. We use the following HMAC:

$$H_{\text{class}}(X, K_{\text{class}}) = \text{HMAC}_{\text{MD5}}(X, K_{\text{class}})$$

Our choice of the HMAC function is a pragmatic one: HMAC-MD5 is relatively fast, still considered secure [10] (in contrast to MD5), and is available through a library already linked by the Objective-C runtime. Note that the choice of HMAC is a security parameter in our defense; we can replace HMAC-MD5 with any stronger (but likely also slower) MAC in case attacks against HMAC-MD5 appear.

The core assumption of our protection scheme is that the attacker does not know the secret key and hence cannot modify the method list or other metadata used during method lookup without being detected. However, metadata may also change for legitimate reasons. Objective-C is a dynamic language which provides APIs for, e.g., adding classes and methods

at run time. We support legitimate changes to metadata via provided APIs by making the change, followed by recomputing the HMAC field.

Note that computing the MAC adds considerable overhead to the slow path lookup (see Section 3.6.2 for empirical evaluation results). However, lookup results are cached so the slow path is only executed once per (class, selector) pair. Therefore, the steady state program performance remains unaffected. This is also reflected in the implementation of the runtime: the fast path consists of hand-tuned assembly code while the slow path is written in C.

3.5.2 Securing the Fast Path

We protect the fast path in a manner similar to the way we secure the slow path. We implement an authentication mechanism for cache entries that detects any tampering. However, in our practical experiments, we have encountered applications, e.g., iTunes, that modify cache entries directly, i.e., writing to the entries in memory instead of using the runtime API, in much the same way an attacker could tamper with the cache. Therefore, we must allow changes to the cache originating outside the runtime and make sure we detect them and fall back to the slow path.

We implement this by extending the fast path lookup algorithm with an additional authentication step for *cache hits*, as illustrated by Figure 3.4 (and with greater detail by Listing 3.4). This additional step computes the MAC of the cache entry and checks it against the MAC stored inside the entry. If the hash matches, the algorithm continues normally. Otherwise, the algorithm considers the authentication failure as a *cache miss*. We also modified the runtime to update the stored MAC on changes to a cache entry. This effectively renders any malicious changes to the cache moot, as they merely decrease program performance with no impact on security as long as the slow path is protected as described in Section 3.5.1.

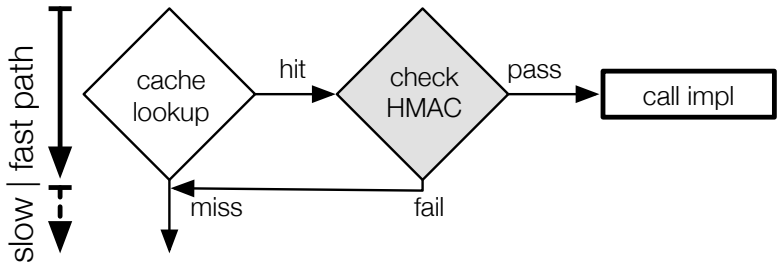

Figure 3.4: Fast path secured with MAC integrity check

Each cache entry contains two pointers: the selector and the method implementation pointer, as shown in Figure 3.3. Using these pointers as the MAC input ensures that the attacker cannot modify existing cache entries or add new ones. However, the attacker could still copy entries between caches for different classes, and we wouldn't be able to detect this. Therefore, we add a third pointer to the MAC input: the pointer to the class that owns the cache. This prevents the attacker from copying valid cache entries between classes, as each cache entry is now associated with a class. Note that the attacker can still remove or invalidate entries from a cache and duplicate cache entries inside the same cache. However, this is harmless since the hash check detects invalid entries and triggers a fallback to the slow path (which is secured separately), while duplicates can be allowed to occur.

Unlike the slow path, performance is critical on the fast path and every additional instruction can have a significant impact. Therefore we selected a MAC that we can implement in as few assembly instructions as possible, and easily integrate into the existing cache lookup algorithm. The NH hash function family used in UMAC [13] meets our performance requirements, so we use a modified version of NH as part of our MAC:

$$H_1(X, K) = \sum_{i=0}^{i \leq 2} (X_L[i] + K_L[i]) * (X_H[i] + K_H[i])$$

where:

- $X = (class, sel, imp)$ is the 192-bit concatenation of the three pointers to hash: the

class pointer, the selector and the method implementation.

- $K = (K_0, K_1, K_2)$ is the 192-bit secret key retrieved from the key store.
- $X_L[i]$, $K_L[i]$, $X_H[i]$, and $K_H[i]$ are the low and high 32-bit words of the i^{th} element of X and K, respectively.

The H_1 function has very low collision probability, but is vulnerable to known plaintext attacks (given a large enough sample of $H_1(X, K)$ outputs and their corresponding X inputs, the attacker can compute the K), and therefore insufficient to use as a MAC. UMAC strengthens NH against these attacks by XORing its result with a random number produced by applying a pseudo-random function (PRF) to a nonce.

Using a strong PRF in this case would take too many processor cycles, however, so we use a faster alternative in the form of a fixed-size table T of random 64-bit numbers. We generate this table at program load time, and store it in memory securely as described in Section 3.5.4. To compute the 64-bit HMAC of a cache entry, we compute H_1, use its output to index into T, and use the resulting value as the output. To simplify indexing, we always allocate T as a table of size $|T| = 2^N$. To compute the index we truncate the output of H_1 to 32 bits and then use the highest N bits. The final form of the HMAC becomes:

$$H_{\text{cache}}(X, K_{\text{cache}}) = T[H_{1[31:(31-N)]}(X, K_{\text{cache}})]$$

Listing 3.4 shows the pseudo code representation of the secured cache lookup algorithm (see Listing 3.2 for the original version of the algorithm).

Listing 3.4: Secured cache lookup algorithm

```
id objc_msgSend(Object obj, Selector sel) {
  class = obj->class;
  cache = class->cache;
  b = cache_hash(sel) % cache->size;
  while (cache->buckets[b] != NULL) {
    if (cache->buckets[b]->sel == sel) {
      // Cache hit
      impl = cache->buckets[b]->impl;
      // Check the HMAC
      (K, T) = get_secrets();
      h1 = H1(class, sel, impl, K);
      h = T[h1 >> (64 - lg(size(T)))];
      if (h != cache->buckets[b]->hmac)
        goto CacheMiss;
      jmp impl;
    }
    b = (b + 1) % cache->size;
  }
CacheMiss:
  // Take the slow path
}
```

3.5.3 Securing the Forwarders

There is another attack vector that the attacker can use during message dispatch: the user forwarder pointers (one for regular message dispatch and one for calls that return structures). The application can legitimately set these pointers using an API call, and many applications use this feature. We prevent attackers from modifying the two pointers by associating a HMAC with each pointer. An alternative is to use a single HMAC over the concatenation of both pointers, however, using two separate HMACs is more efficient. The runtime updates the HMAC whenever it changes one of the pointers, and checks the HMAC before calling any of the handlers. We once again use a helper function:

$$H_2(fwd, K) = (fwd_L + K_L) * (fwd_H + K_H)$$

that combined with the table T gives us the HMAC:

$$H_{\text{fwd}}(fwd, K_{\text{fwd}}) = T[H_{2[31:(31-N)]}(fwd, K_{\text{fwd}})]$$

3.5.4 Secure Key Store

Our defense must keep several pieces of information secret to attackers: the HMAC keys—K_{class}, K_{cache}, and K_{fwd}—and the random table T. Discovering these values would allow the attacker to forge the HMAC values and bypass our defenses. It is therefore critical that we prevent attackers from disclosing or guessing these values.

To hide secrets from attackers, we rely on a security primitive known as XoM. This construct allows us to map virtual memory pages in memory so that they will generate a segmentation violation if accessed by anything other than the CPUs instruction fetch unit. Embedding secret values inside such pages allows the runtime to retrieve the values using function calls, while at the same time preventing attackers from reading the pages using direct information leaks. As outlined in Section 3.3, our threat model assume that one of the available XoM implementations [9, 14, 21, 35] has been deployed on the Objective-C runtime.

We store every secret value inside an execute-only accessor function that returns that secret value in a register when called (the value itself is embedded in the body of the accessor). Additionally, the attacker cannot call the accessor, since that would require hijacking the control flow of the program.

Using one accessor per 64-bit secret value would increase memory usage significantly (we would need an 11-byte accessor for every 8-byte secret, producing a memory overhead of 37.5 %), so we take another approach. We store the keys along with T inside a read-only memory region allocated at a random memory address (chosen randomly when calling `mmap()`),

then store a pointer to this region inside an accessor. To access the table, the runtime calls the accessor to get the pointer, then accesses T using a regular memory read.

To simplify our implementation and reduce the number of accessor calls, we store the HMAC keys as extra cells (one per every 64 bits of key) inside the table T and perform a single accessor call to get the keys and table pointer. This lets the runtime retrieve all secret values using a single accessor call on the fast path, as opposed to one or more per key and then one for the table.

3.6 Evaluation

In the following, we discuss evaluation results related to the security and performance of our proposed defense.

3.6.1 Security

We evaluated the effectiveness of our defense using the proof-of-concept Subversive-C exploit described in Section 3.4. Our hardened runtime is a drop-in replacement which lets us keep all other parameters the same. Thus we can be confident that any differences during program execution are caused by our defense. When running our original attack without any adaptations, the program terminates either due to failing pointer checks (in most cases) or integrity checks. The reason for this is that our original attack does not generate all data structures touched by the integrity checks, but rather the bare minimum necessary to exploit message dispatch. Therefore some dereferenced pointers stay uninitialized. Even if an "accidental pointer" references valid memory, the actual integrity check fails due to the mismatch between computed and stored values.

Next, we extended our attack to generate all data structures that are needed for metadata verification, i.e., all structures that act as input for the HMAC. The easiest way to do so is to copy and then modify existing class structures. However, we were unable to compute the correct HMAC values used to secure the contents of both the cache and the method list. This left us with guessing the right value as the only remaining choice, which is difficult since we need to guess correctly for every counterfeit object in the chain. An incorrect guess for any object leads to detection and program termination.

In both cases Subversive-C is detected before any attacker-controlled code is executed. Even the initial object can not be dispatched. More specifically, program execution is aborted on the first message that is dispatched to a counterfeit object. As expected, we can create (valid) empty caches, or use the fallback mechanism of the cache protection which triggers a slow path lookup whenever the cache integrity check fails. Creating valid cache entries is difficult due to the keys used in the HMAC being inaccessible to the attacker. With the cache secured, we can try to forge the HMAC for the method list. Here we face even stronger security guarantees since we need to forge HMAC-MD5. Again the attacker lacks the knowledge of the input keys which are protected by the secure key store. Both the cache and the method list protections can only be circumvented by guessing the correct values, which has a very low success rate.

The third way to gain code execution would be to overwrite the forwarders. However, even with an arbitrary write primitive to allow modifications of these handlers, this will not work. They are protected and the attacker again lacks the secret keys to generate valid handler entries.

We therefore conclude that our hardened runtime probabilistically detects and prevents Subversive-C exploits.

3.6.2 Performance

Since there is no standard set of Objective-C benchmarks, we compiled the following list of programs to evaluate the performance of our hardened runtime:

- **Dispatch** (micro) invokes a dynamically dispatched (and empty) method in a tight loop. The empty method is invoked 10^8 times.
- **Fibonacci** (micro) computes the 35th Fibonacci number using naive recursion.
- **Sorts**[4] (micro) uses different sorting algorithms (merge, quick, bubble, heap, insertion, selection, and the Objective-C library sort) to sort integer arrays of size 10^4. We combine the running times of all algorithms for our purpose.
- **XML parser**[5] (application) parses and creates song objects from XML data (100 or 1000 entries) using the `NSXMLParser` class [6] from the Objective-C standard library.
- **iTunes play** (application) plays a 5 second audio clip and closes iTunes.
- **iTunes encode** (application) converts a 4 minute song in MP3 format (7 MB) to M4A (7.6 MB) using the AAC encoder provided by iTunes.
- **Pages PDF** (application) exports a 100 page document (270 KB) to PDF (327 KB) in Pages (Apple's word processor).

When reporting results we average over 100 and 10 runs for micro benchmarks and application benchmarks, respectively. We automate the application benchmarks using AppleScript [5] which increases the consistency of our results and allows us to interact with real-world applications. Our hardened runtime is based on the Objective-C runtime version 532.2

[4] The Sorts benchmark [47] was developed by Jesse Squires.

[5] XML parser is an adaptation of a benchmark from Apple [7] that compares the performance of XML parsing libraries on iOS.

Table 3.2: Micro benchmark `msgSend` invocation counts and overheads

Benchmark	`msgSend` calls	Calls/ms	Overhead
Dispatch	10,000,000,215	190,583	106.46 %
Fibonacci	2,986,070,515	173,527	88.66 %
Sorts	13,329,480,611	82,597	34.54 %
		148,902	76.55 %

Table 3.3: Application benchmark `msgSend` invocation counts and overheads

Benchmark	`msgSend` calls	Calls/ms	Overhead
XML-100	7,940,898	6,475	2.81 %
XML-1000	78,119,698	6,386	1.97 %
iTunes play	8,592,257	1,667	0.37 %
iTunes enc.	114,948	29	1.82 %
Pages PDF	78,691	46	0.75 %
		2,921	1.54 %

(x86-64), which we use as the baseline for performance comparison. Experiments were conducted on an iMac 2.8 GHz Intel Core i7 with 8 GB memory running OS X Yosemite (10.10.5) and the latest versions of iTunes and Pages. In addition, we ran each benchmark with an instrumented version of our runtime to count the number of times the general dispatch function `msgSend` is invoked. Table 3.2 and 3.3 report the results of our experiments. Note that the reported numbers do not include the overhead for the defenses assumed in Section 3.3.

The goal of the Dispatch benchmark is to give us an upper bound for the overhead incurred by our hardened runtime. This is the case since the benchmark does no real work and just calls an empty method repeatedly. This puts maximum pressure onto the message dispatch mechanism which is the only part of the runtime affected by our protection scheme. Using the data from Table 3.2 we conclude that the maximum slowdown is bounded by $2x$.

The Fibonacci benchmark mainly executes recursive method calls plus an add operation and some control flow to terminate recursion. Note that we mean dynamically dispatched call, i.e., calls dispatched via `msgSend`, whenever we write method call in this section. Standard C function calls are valid in Objective-C, but do not go through `msgSend`. Therefore our

Table 3.4: Startup times and overhead in milliseconds

Benchmark		HelloWorld	iTunes
Startup	Base	35	1020
	Hardened	107	1478
	Total	**72**	**458**
Overhead	Random table	43	43
	Integrity checks	29	415

defense does not reduce the performance of regular calls to C functions.

The Sorts benchmark is implemented in a way that leads to a high number of `msgSend` calls. Rather than using plain integer arrays, it uses Objective-C collections that require boxing of the integer numbers they store. So what normally is a simple array access becomes two method calls: one to index the collection and one to unbox the integer for comparison. The benchmark results reflect this accordingly. To back our claim we modified the benchmark to use plain integer arrays, replacing `NSMutableArray` with (`int arr[], int len`). As expected, the difference in running times then falls into the range of measurement noise ($< 1\%$).

At this point we want to draw attention to the relation between `msgSend` calls per millisecond and the reported overhead. For compute-intensive programs it is directly proportional. In other words: the more real work a program does, the smaller the overhead.

With the second set of benchmarks we want to demonstrate that although overhead for individual micro benchmarks is considerable, it is insignificant in practice. Especially for interactive applications like iTunes and Pages there is no perceivable slowdown during normal use. For the benchmarks iTunes play and Pages PDF the reported overhead is in the range of measurement noise. Our explanation is that Objective-C is mostly used to implement an application's logic and user interface while core functionality (playing and encoding music files, exporting to PDF) is provided by C libraries. Hence, we incur little to no overhead on those activities. The only time an end user experiences additional delay is during program startup. Table 3.4 quantifies this delay.

We measured the running time of a simple HelloWorld program and the startup time of iTunes both with our baseline and hardened runtime. The total startup overhead for HelloWorld is 72 ms, whereof 43 ms are attributed to seeding the random table which aids the implementation of the secure key store. The remaining 29 ms are spent to populate and check hashes of 280 core classes, e.g., `NSObject`, which are eagerly initialized by the runtime. The time needed to seed the random table depends linearly on the size of the table. In our implementation the table holds 1 million keys resulting in 8 MB total size. The size of the table can be adjusted to adhere to an application's security and memory constraints.

For iTunes the total startup overhead is about half a second. This is due to iTunes being a complex application initializing roughly 2000 Objective-C classes during startup. Considering typical application usage patterns we argue that this is acceptable since there is no further perceived slowdown during continued use.

3.7 Discussion

Type Confusion Our changes to the Objective-C runtime secure it against attackers changing existing classes or adding new ones. However, attackers can still perform so-called *type confusion* attacks, where attackers would change the type of an object to another legitimate type. Extending the MAC-based defenses in this work to the objects themselves, as opposed to just classes, could prove to have a significant performance impact. It would require storing a MAC in every object and verifying an object's class pointer every time a method is called. We leave the analysis of this impact, as well as the discovery of efficient alternatives, as future work.

Secure Key Store Attack Surface In Section 3.5.4, we presented our approach to securing the key store against leaks: we store its contents at a random address in memory,

then store the address as a pointer inside XoM. Since the pointer is stored in a single non-readable location in memory, attackers cannot use an information leak attack to locate the table. However, this approach could expose the table to attackers in some other way, e.g., probing all memory pages one by one to find the table. However, probing attacks would be difficult for two reasons. First, locating all readable virtual memory pages is difficult, assuming attackers cannot install a signal handler or obtain a virtual memory map for the program. Second, to identify the table T in memory, attackers would need to distinguish between randomly-generated bytes and proper program data. Therefore, the barriers to attackers locating T are high. Choosing whether to store the random table in execute-only or readable memory presents a potential security vs. memory usage trade-off. Storing T directly in XoM provides guaranteed secrecy, at the cost of an extra 37.5 % memory usage for the table. We therefore leave this decision to system developers.

Side-channel Attacks Side-channel attacks are another potential class of attacks against the key store, or more specifically against the table T. For example, attackers could derive the indices used to access the table, and therefore the values of H_1, by measuring the externally-visible metrics (such as cache misses or CPU cycles) while the runtime performs its integrity checks (similar attacks have been demonstrated on cryptographic functions [96]). If such attacks prove feasible and likely, the same defenses that protect cryptographic algorithms can also be applied to our key store [20].

Object Layout Randomization One other interesting mitigation is object layout randomization. In the runtime, the offsets of instance variables from the start of an object are dynamically defined when its class is loaded. The Objective-C language puts no constraints on the order of variables inside an object, i.e., there is no requirement that they be in the same order that they appear in the source code. Therefore, it would be possible to randomize the object layout. This would not defend against an attacker who can read all of memory,

but would make it harder to inject counterfeit objects.

3.8 Conclusion

We present Subversive-C, the first whole-function reuse attack abusing the `msgSend` feature of Objective-C. Our attack shows that COOP-style attacks which are far harder to prevent than ROP-style code-reuse, are not limited to C++ code. We discuss the intricacies of Objective-C message dispatch and how to utilize them for our attack. Specifically, we describe an attack targeting the AppKit library (x86-64) for OS X, which is a core building block for many popular applications. Finally, we present a practical defense against Subversive-C and show that it imposes a negligible performance overhead when protecting real-world applications.

Chapter 4

Fast and Flexible Sanitization via Run-time Partitioning

4.1 Motivation

Although modern, safe languages could gradually replace C/C++, the sheer amount of legacy systems code forces security researchers to search for and fix memory corruption vulnerabilities in existing code in the near term. While some bugs can be found through static program analysis, many cannot. Sanitizers are dynamic analysis tools that can detect memory corruption and many other problems as well as pinpoint their occurrence during program execution [59, 79, 88]. To increase coverage, sanitizer runs can be driven by a fuzzer. A fuzzer simply feeds the program random inputs and records inputs that generate crashes or cause previously unexecuted code to run.

Sanitizers instrument programs—usually during compilation—to detect issues such as memory corruption and undefined behavior. This instrumentation incurs significant overheads, so sanitizers are turned off in release builds and traditionally only enabled on internal quality

assurance builds that run on high-end hardware. This is less than ideal as the number of paths executed by test suites and fuzzers is outnumbered by the number of paths executed by end users.

In a recent experiment, the Tor Project released sanitizer-enabled (labeled "hardened") builds directly to its users [48]. The hardened build series was discontinued in part due to the high performance overhead and in part due to confusion among end users over which version to download. With access to PartiSan, the Tor Project developers could have released builds that automatically adapt the level of sanitization to the capabilities of the host system. Overhead can be limited by using a conservatively low, adaptive threshold by default (and possibly disabling sanitization completely on underpowered systems) while simultaneously allowing expert users to modify the default settings (thereby also eliminating the need for multiple build versions).

This dissertation presents PartiSan, which clones frequently executed functions at compile time and efficiently switches among them at run time. Each function variant can be optimized and sanitized independently, and thus has different security and performance properties. In the simplest case, one variant is instrumented to sanitize memory accesses while the other one is not. PartiSan supports configurable run-time partitioning policies that determine which variant is invoked when a function is called. For example, PartiSan can execute slow variants (e.g., variants with expensive checks) with low probability on frequently executed code paths, and with high probability on rarely executed paths. This policy helps us keep the sanitization overhead below a given threshold.

This is superficially similar to the ASAP framework by Wagner et al. [100] insofar that both approaches explore the idea of reducing the amount of sanitization on the hot path. However, ASAP *statically* partitions the code into parts with or without sanitization based on previous profiling runs at compile time. PartiSan prepares programs for partitioning at compile time but does the partitioning *dynamically* at run time. This allows us to produce a single binary

Table 4.1: Conceptual comparison of ASAP and PartiSan

Statement	ASAP	PartiSan
Goal ...	deploy sanitizers as mitigations	find bugs efficiently
Partitioning is ...	static (compile time)	dynamic (run time)
Overhead reduction ...	removal of expensive checks	probabilistic checking
Profile data is ...	required	beneficial
Code is ...	deleted	cloned
Assertions are ...	removed	retained
Detect bugs in cold code ...	always	always
Detect bugs in hot code ...	never	probabilistically

that adapts to each individual host system, sanitizing as many paths as possible under a given performance budget. Moreover, we can create N different function variants to support $N - 1$ types of sanitization in a single binary. Table 4.1 contrasts PartiSan and ASAP.

Both our work and ASAP build on the assumption that security vulnerabilities in frequently executed code get discovered and patched relatively quickly, whereas vulnerabilities in rarely executed code might go unpatched for a long time. Applying sanitizer checks only to rarely executed code therefore limits the sanitizer's performance impact, while maintaining most of its effectiveness. Contrary to ASAP, however, PartiSan never fully removes checks. Instead, PartiSan lowers the probability that checks in frequently executed code get executed.

4.2 Background

Sanitization has become an active research topic in the past few years. LLVM [54], the premier open-source compiler, includes five different sanitizers. We demonstrate PartiSan by applying two of these sanitizers to a variety of programs. ASan, short for AddressSanitizer [79], instruments memory accesses and allocation operations to detect a range of memory errors, including spatial memory errors such as out-of-bounds accesses and temporal violations such as use-after-free bugs. UBSan, short for UndefinedBehaviorSanitizer [59],

currently detects 22 types of operations whose semantics are undefined [56] by the C standard [45]. UBSan includes checks for integer overflows, uses of uninitialized or unaligned pointers, and undefined integer shifts.

We used these sanitizers with PartiSan for two reasons. First, the combination of ASan and UBSan detects many of the vulnerabilities that are security critical. Second, both sanitizers can be applied selectively.[1] Removing any of the sanitization checks from a program does not affect the correct functioning of the remaining checks. This makes these sanitizers a good fit for our framework, in which we selectively skip sanitization through run-time partitioning.

Analogous to previous work [30, 79, 100], we do not apply PartiSan to JIT code to reduce the required engineering effort. However, we do not anticipate fundamental issues for extending our technique to JIT-compiled code.

4.3 PartiSan

We start with a conceptual overview of PartiSan, and then discuss each component in depth.

4.3.1 Overview

Our goal is to reduce the run-time overhead of the sanitizers. We do this by creating multiple variants of each function, applying sanitizers to some variants, and embedding a runtime component that partitions the execution of the program into sanitized/unsanitized slices based on a policy.

Figure 4.1 shows an overview of the PartiSan system. To apply PartiSan to an application,

[1] Note that ASan requires metadata to execute checks. The maintenance of this metadata constitutes residual overhead which cannot be removed.

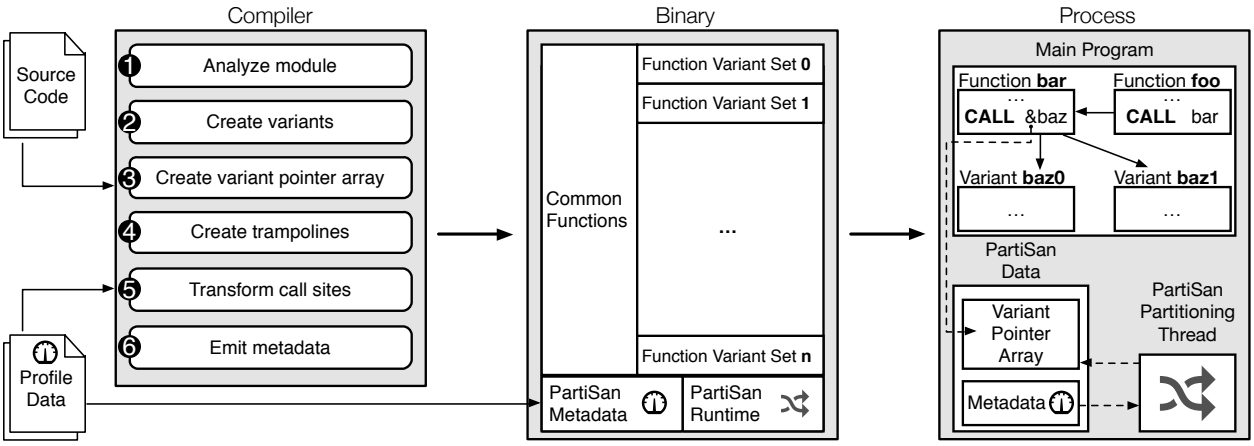

Figure 4.1: System overview. The compiler (left) creates PartiSan-enabled applications (center) that have multiple variants of each function. A run-time indirection through the variant pointer array (right) ensures that the control flow calls the currently active variant. PartiSan's runtime periodically activates function variants according to the configured partitioning policy.

the developer must compile the source code of the program with our modified compiler (left side of Fig. 4.1). Some partitioning policies require that the developer supply profile data. Sanitizers and the partitioning policy are selected through the compiler's command line arguments.

The compiler generates an application with multiple variants for each function. To simplify the following discussion, we will focus on use cases where we generate two variants. One of the variants, which we refer to as the *unsanitized variant*, does not include any sanitizer checks. The other variant, which we call the *sanitized variant*, incorporates all sanitizer instrumentation.

The compiler modifies the program's control flow as follows. Rather than calling functions directly, the functions call each other through an additional level of indirection. Specifically, the compiler embeds a "variant pointer array" containing one slot for each function in the program code. At run time, each slot holds the pointer to the currently active variant of the corresponding function. The PartiSan runtime, which is linked into the application by our compiler, periodically selects and activates one variant of each function according to the

configured partitioning policy. This layer of indirection ensures that a call always targets the *active variant* of a function.

The runtime comes with three predefined partitioning policies: random partitioning, profile-guided partitioning, and expected-cost partitioning. With the random partitioning policy, the runtime randomly selects the active variants, whereas the profile-guided and expected-cost partitioning policies select active variants with a probability that depends on the execution frequency ("hotness") and/or expected sanitization cost of that function. These policies can help us limit the cost of sanitization.

4.3.2 Creating Function Variants

PartiSan's compiler pass runs after the source code is parsed and converted into intermediate representation (IR) code. As its first step (Step ① in Figure 4.1), our compiler pass analyzes the IR code and determines which functions to create variants for. We do not necessarily create multiple variants for each function. If the developer selects the profile-guided or expected-cost partitioning policy, and if the profile data indicates that a function is infrequently executed, then we create only the sanitized variant for that function. This design choice prevents PartiSan from unnecessarily inflating the code size of the program and is justified because checks in infrequently executed code have little impact on the program's overall performance.

Then, PartiSan creates the function variants (②). First, we clone functions that should have two variants and give them new, unique names. Then, we apply the requested instrumentations to the variants. Depending on which instrumentation should be applied, we have two strategies to apply transformations:

Instrument only to the sanitized variants This is our default and preferred strategy. It is only available for sanitizers that can be applied on a per-function basis. One example of such a protection is ASan, which can be applied to functions by marking them with a function-level IR attribute.

Instrument all variants, then remove it from the unsanitized variant We identified two cases that necessitate this strategy. First, some sanitizers add their instrumentation code early in the compiler pipeline or even in the front end (e.g., UBSan). Second, it can be used to apply sanitizers that maintain metadata since removing the functionality that maintains metadata from the unsanitized variants might break the sanitizer checks performed in the sanitized variants. When using this strategy, we instrument code uniformly across the entire translation unit, and then run a secondary pass later in the compilation pipeline to remove the checks from the unsanitized variants.

4.3.3 Creating the Indirection Layer

Once the function variants are created, our compiler pass creates the indirection layer, through which we route all of the program's function calls. This ensures that the program can only call the active variant of each function. Our indirection layer consists of three components: the variant pointer array (right side of Figure 4.1), trampolines, and control-flow instructions that read their target from the pointer array.

Our compiler starts by embedding the variant pointer array into the application (③). The pointer array contains one slot for each function that has multiple variants. Each slot contains a pointer to the entry point of the currently active variant of that function.

Then, we create trampolines for externally reachable and address-taken functions (④). A trampoline utilizes a tail call to efficiently jump to the currently active variant of its asso-

ciated function. We assign the original name of the associated function to the trampoline. This way, we ensure that any call that targets the original function now calls the trampoline, and consequently, the currently active variant of the original function instead.

Finally, we transform all direct call instructions that target functions with multiple variants into indirect control-flow instructions that read the pointer to the active variant of the target function from the pointer array (⑤). This optimization eliminates the need to route direct calls within the program through the function trampolines. However, the trampolines may still be called through indirect call instructions, or by external code.

4.3.4 Embedding Metadata

Our compiler embeds read-only metadata describing each function and its variants into the application (⑥). The metadata typically consists of the function execution frequencies read from the profile data, the estimated execution costs for all function variants, and information connecting each slot in the variant pointer array to the variant entry points associated with that slot. The partitioning policies can use this embedded metadata to inform their run-time decisions.

4.3.5 The PartiSan Runtime

Our runtime implements the selected partitioning policy by activating and deactivating variants. While a specific variant is active, none of the other variants of that same function can be called. To activate a variant, our runtime writes a pointer to that variant's entry point into the appropriate slot in the pointer array. PartiSan periodically activates variants on a background thread. This allows developers to employ a variety of partitioning policies without slowing down the application thread(s). Operating on a background thread also

allows our runtime to run frequently, and thus make fine-grained partitioning decisions. Our runtime currently implements the following three partitioning policies.

Random Partitioning With this policy, our runtime component activates a randomly selected variant of each function whenever our thread wakes up. Since we only generate two variants of each function, this policy divides the execution time evenly among the sanitized/unsanitized function variants.

Profile-Guided Partitioning With this policy, our runtime component collects the list of functions with multiple variants in the program and orders this list based on the functions' execution counts recorded during profiling. Our runtime activates the sanitized variant of a function with a probability that is inversely proportional to its order in the execution count list. The sanitized variant of the most frequently executed function is activated with 1% probability, and that of the least frequently executed function with a 100% probability. Note that this partitioning policy does not estimate the overhead impact of executing a sanitized variant instead of an unsanitized variant. It also does not consider the absolute execution count of a function. For example, the second least executed function in a program with 100 functions is sanitized 99% of the time, even if its execution count is 1000 times higher than that of the least executed function.

Expected-Cost Partitioning This policy improves upon the profile-guided partitioning policy by calculating sanitization probabilities based on function execution counts (read from the profile data) and estimated sanitization cost. We estimate the cost of sanitization for each function by calculating the costs of all function variants using LLVM's Cost Model Analysis [73] and weigh each instruction's cost with its relative execution count. We then

calculate the probability of activating the sanitized variant for a function using formula:

$$P_{sanitization}(f) = \frac{sanitization\,budget(f)}{cost_{sanitization}(f) * execution\,count(f)}$$

The sanitization overhead budget is chosen by the developer and is evenly distributed among the functions in the program.

In addition to these three predefined partitioning policies, developers may define their own. Policy implementations can separately define the *static* and *dynamic* aspects of partitioning. On program startup, the static part of a partitioning policy computes the variant activation probabilities based on the embedded function metadata and—if available—collected profile data. Afterwards, while the program is running, the dynamic part uses the computed probabilities in combination with other runtime data to decide which function variants to activate. This additional runtime data can be general (e.g., adaptive overhead threshold based on system load) or application-specific (e.g., differential treatment of web server requests based on their origin). For our fuzzing use case we implemented a custom partitioning policy which forgoes static profile data in favor of dynamic coverage data. Section 4.8 elaborates on the mechanics of implementing a custom policy.

4.4 Implementation

Our prototype implementation of PartiSan supports applications compiled with Clang/LLVM 5.0 [54] on the x86-64 architecture. Our design, however, is fully generalizable to other compilers and architectures. The prototype consists of two main components: a compiler pass which creates function variants, the indirection layer, and function variant metadata; and a runtime which continuously partitions control-flow according to the developer-selected policy.

4.4.1 Profiling

Two of our run-time partitioning policies rely on profile data to calculate the sanitization probabilities. We use LLVM's built-in profiling functionality to generate binaries that collect profile data.

4.4.2 Compiler Pass

Our pass instruments the program code at the LLVM IR level processing one translation unit at a time. The applied instrumentations preserve ABI semantics, therefore, PartiSan-enabled modules are compatible with standard code without recompilation. In addition, the PartiSan compiler itself is fully compatible with standard build systems and program loaders. We scheduled our pass to run right before the LLVM sanitizer passes, which run late in the compiler pipeline. This allows us to define (mostly declaratively) which variants get instrumented without interfering with LLVM's earlier optimization stages.

Creating Function Variants Of the sanitizers bundled with LLVM, our pass currently supports ASan and UBSan. We did not modify any sanitizer code and most of PartiSan's code is tool-agnostic. To create the sanitized function variants, we pass the necessary `-fsanitize` command line options to the compiler. PartiSan then creates copies of these sanitized functions and removes their instrumentation to obtain their unsanitized counterparts. ASan's front-end pass prepares the program by marking all functions that require sanitization with a function-level attribute. With just one line of ASan-specific code, PartiSan removes this function attribute for the unsanitized variants. UBSan's front-end pass embeds many of its checks before the program is translated into IR. PartiSan contains 45 lines of code to remove these checks from the unsanitized variants. PartiSan also marks all created function variants and trampolines with attributes that identify them as such.

Creating the Indirection Layer PartiSan routes function calls through an indirection layer. This layer ensures that the program can only call the active variant of each function. We create the indirection layer as follows. We begin by collecting the set of functions that have multiple variants. Then, we add the variant pointer array as a global variable with internal linkage. We choose the size of the array such that it has one slot for every function in the set. Next, we create trampolines for all functions in the set. The trampoline, which takes over the name of the function it corresponds to, forwards control to the currently active variant of that function. By taking over the name of the original function, the trampoline ensures that any calls to that function will be routed to the currently active variant.

Next, we replace all call instructions that target functions in the set with indirect call instructions that read their call target from the variant pointer array. Functions outside the compilation unit will not be in the set, but might still have multiple variants. While we do not replace calls to such instructions, the call will still (correctly) invoke the currently active variant of the target function because it will be routed through that function's trampoline. The usage of trampolines also ensures compatibility between PartiSan-enabled and uninstrumented modules.

Note that the compiled program will only contain the trampolines that may actually be used at run time. If a trampoline's corresponding function is not externally visible (and thus cannot be called by external code) and it does not have its address taken (and thus cannot be called indirectly), then the trampoline will be deleted by LLVM's dead code elimination pass.

Embedding Metadata Our runtime component needs to know which function variants are associated with each slot of the variant pointer array. Depending on the partitioning policy, it may also require function execution frequencies and estimated execution costs for all function variants. We add this information as read-only data to each compilation unit.

Listing 4.1: Function descriptor definition

```
typedef struct {
  const uint32_t v_count;      // Number of variants
  const uintptr_t* variants;   // Variant pointers
  const int64_t* rel_costs;    // Variant cost relative to baseline
  const uint64_t exec_count;   // Execution count during profiling
  uint32_t* probs;             // Activation probabilities for each
                               ↪ variant (determined by policy)
} func_t;
```

Specifically, we add an array of function descriptors to each unit. The definition of a function descriptor is shown in Listing 4.1. The layout of the function descriptor array mirrors that of the variant pointer array.

Since we apply PartiSan to individual compilation units, there might be multiple variant pointer arrays and function descriptor arrays in the program. Rather than forcing our runtime to find the locations of all of these arrays, we explicitly pass the array locations to the runtime during program startup. We do this by generating a constructor function for each module. We add this constructor function to the binary's `.init_array` section, so it is automatically invoked when the program interpreter loads our program. The constructor function is responsible for registering the module with our runtime.

4.4.3 The PartiSan Runtime

The PartiSan runtime implements the three partitioning policies described in Section 4.3.5. The runtime is linked into the final executable and exposes a single externally visible function used to register modules: `cf_register(const func_t* start, const func_t* end)`. Every module registers its function variants with the runtime by invoking this function from a constructor. After all modules have registered, the runtime initializes. To initialize the runtime, we add yet another constructor function to the program. This constructor initializes the runtime's data structures and starts the partitioning background thread. We assign the

lowest possible priority to the runtime's constructor, so it is guaranteed to run after all of the modules have registered.

The runtime's initialization proceeds in four steps. First, the runtime computes the activation probabilities for each function variant, according to the configured policy. Depending on which policy the runtime enforces, these activation probabilities may be based on the function execution counts, expected variant costs, and additional dynamic properties. Then, we seed a secure number generator. Next, we initialize all variant pointer arrays. This is necessary because the program might call some of the variant functions before our runtime's background thread performs its first round of run-time partitioning. Finally, we spawn the background thread that is responsible for the continuous run-time partitioning. The background thread runs an infinite loop which repeatedly invokes the partitioning function, followed by a call to `nanosleep()`. The sleep time can be configured via an environment variable and defaults to 100 ns.

Run-time Partitioning Our background thread runs an infinite loop, which invokes the partitioning procedure (shown in Listing 4.2) whenever it wakes up. This procedure iterates through the function descriptors for every registered module (lines 3–5). For every function, we generate a random integer number X between 0 and 100, and use this to select one of the variants (`pick_variant()` in line 6). If the activation probabilities for the sanitized and unsanitized variants of a function are 0.01 and 0.99, respectively, then we will activate the sanitized variant if X is less than 2, and we will activate the unsanitized variant for values greather than 1. Finally, we write the pointer to the activated variant in the variant pointer array (line 10). Note that the developer can provider her own `pick_variant()` function to customize the dynamic aspect of the partitioning policy.

We attempt to reduce cache contention by performing the write only if necessary (i.e., only if the old and new value differ). This adds a read dependency on the old pointer value which

Listing 4.2: Variant partitioning procedure

```
1  void partition_variants(module_t* modules, uint32_t m_count) {
2    size_t m, f;
3    for (m = 0; m < m_count; m++) {
4      for (f = 0; f < modules[m].f_count; f++) {
5        func_t*    fn  = &modules[m].functions[f];
6        size_t     v   = pick_variant(fn);
7        uintptr_t* loc = &modules[m].rand_ptrs[f];
8        uintptr_t  val = fn->variants[v];
9        if (*loc != val)
10         *loc = val;
11     }
12   }
13 }
```

```
foo:
  ...
  # Prepare arguments
  callq bar
  ...
```

(a) Original call site

```
foo_0:
  ...
  # Prepare arguments
  callq *.Lptr_array+16(%rip)
  ...
```

(b) Transformed call site

```
bar:
  # Preserve arguments
  jmpq *.Lptr_array+16(%rip)
```

(c) Trampoline

Figure 4.2: Generated x86-64 machine code

may slow down the background thread. However, the execution of the background thread is not performance critical since it runs fully asynchronously with respect to the application threads.

4.4.4 Architecture-Specific Considerations

Our design and implementation are general, however, in our evaluation we focus on Linux running on the x86-64 architecture. Therefore we manually verified the quality of the resulting machine code for this configuration. Figure 4.2 shows the generated x86-64 machine code for (a) an original call site, (b) a transformed call site, and (c) a trampoline.

In the original program, the function `foo` invokes `bar` via a direct call. In the PartiSan-

Table 4.2: PartiSan configuration options

#	Component	Option	Possible values (**default**)
1	compiler	`enable-partisan`	N/A
2	compiler	`fprofile-instr-use`	Path to the file containing profile data
3	compiler	`fsanitize`	Any supported sanitizer
4	compiler	`min-exec-count`	Integer (**10**)
5	compiler	`create-variants-by-hotness`	`all`, **`ignore-cold`**, `only-hot`
6	compiler	`create-variants-by-memory-access`	`all`, **`ignore-no-access`**, `ignore-read-only`
7	runtime	`load-policy-func`	`random`, `profile-guided`, **`cost-model`**
8	runtime	`pick-variant-func`	`random`, `profile-guided`, **`cost-model`**
9	runtime	`thread-sleep-time`	Integer (**100**)

enabled program, `foo_0` (a variant of `foo`) invokes one of the variants of `bar` by loading the appropriate pointer from the variant pointer array and performing an indirect call. The offset `+16` means that the variant pointer for the function `bar` is located in the third slot (at index 2) of the variant pointer array.

The third listing depicts the entire code of a trampoline for the original function `bar`. The trampoline code does not touch any registers or stack memory. This means that the caller's arguments are left intact, including arguments for variadic functions like `printf()`. This simple construction forwards the call through the variant pointer array analogous to the transformed call site. The only difference is that we use a `jmpq` instead of a `callq` instruction, which is an instantiation of tail call optimization [19]. Note that a trampoline is only created if the linkage type of the original function necessitates it (externally visible) or the original function has its address taken.

4.4.5 PartiSan Configuration

PartiSan provides a number of command line options to configure the compiler pass and runtime. Table 4.2 lists the supported options.

PartiSan's configuration options allow the developer to fine-tune the predefined policies

or create completely new ones on a per-application basis. The first three options enable PartiSan, select the sanitizers to apply, and specify the location of the profile data. Options 2 and 3 are standard Clang/LLVM options.

The next three options determine for which functions our compiler creates variants. Option 4 and 5 depend on the function execution counts recorded in the profile data. Option 4 ensures that we only create variants for functions with an execution count of at least the specified minimum. For Option 5 we use LLVM's built-in cost model to classify functions as cold, normal, or hot. We then give the developer the option of creating variants of all functions, of all functions except those that are cold, and of only the hot functions. Similarly, Option 6 distinguishes functions based on how they access memory. This becomes important when we consider PartiSan's interaction with different sanitizers. For example, when using ASan it makes little sense to create variants for a function that does not access memory.

The next two options (7 and 8) allow us to specify the runtime functions which define the static (compute variant activation probabilities) and dynamic (decide which variant to activate) aspects of the partitioning policy, respectively. These options allow us to implement all three of our partitioning policies within the same runtime library. Developers may also define their own custom policy by linking in replacements for the two aforementioned functions.

Lastly, Option 9 configures the delay between rounds of partitioning on the background thread (in nanoseconds).

For the SPEC benchmark suite, we found the performance impact of the aforementioned configuration options to be less pronounced than initially expected. Our explanation is that for SPEC programs the fraction of hot code is particularly small and therefore—since cold code does not affect performance—we get stable performance as long as we avoid making bad decisions for hot code. However, we were able to reduce the size of program binaries by using more restrictive values for options 4 and 5. Unless stated otherwise, the security and

Table 4.3: Evaluated CVEs

CVE #	Program (Submodule)	Vulnerability	Sanitizer	Detection
2016-6297	Php 7.0.3 (Zip extension)	Integer ovf. → Stack ovf.	UBSan, ASan	71.8 %
2016-6289	Php 7.0.3 (Core engine)	Integer ovf. → Stack ovf.	UBSan, ASan	Always
2016-3191	Php 7.0.3 (Pcre extension)	Stack overflow	ASan	6.2 %
2014-0160	OpenSSL 1.0.1f (Heartbeat ext.)	Heap over-read	ASan	Always
2014-7185	Python 2.7.7 (Core library)	Integer ovf. → Heap over-read	UBSan	Always

performance evaluations in the following sections use the default values from Table 4.2.

4.5 Effectiveness

We evaluate the effectiveness of PartiSan with an empirical investigation of five CVEs [69], including the infamous Heartbleed bug [31]. Table 4.3 shows the CVEs we tested. Each of them was found in a popular real-world program and the types of vulnerabilities include stack-based overflows and information leaks on the heap. We used PartiSan to compile two versions of each program, applying ASan to the sanitized variants in one version and UBSan in the other version, and we configured our runtime to enforce its expected-cost partitioning policy. We detected four out of five vulnerabilities with the ASan version, and three out of five with the UBSan version. We then compiled a third version of the program with the same partitioning policy and applied both sanitizers to the sanitized variants. This third version reliably detects three out of five CVEs. The remaining two CVEs are detected in 72 % and 6 % of our test runs.

For each of the selected CVEs we perform the following steps:

1. Verify vulnerability exposure
2. Verify vulnerability detection
3. Collect profile data

4. Evaluate vulnerability detection with PartiSan

Each of the above steps requires a program version with different instrumentation. In step 1, we compile the vulnerable program without any instrumentation and verify that the vulnerability can be triggered. To do this, we adapt the proof-of-concept scripts referenced in the CVE details.

In step 2, we compile the program with ASan or UBSan enabled, but without PartiSan. We run our test script from step 1 to verify that the vulnerability is detected by the sanitizer.

Our expected-cost partitioning policy greatly benefits from profile data, so in step 3, we use LLVM's built-in profiling facilities to create an instrumented version of the program for collecting profile data. We use the tests that come with the program as the profiling workload. For vulnerabilities in submodules/extensions, we only run the tests of the submodule to increase the chance of the vulnerable code being classified as hot (since vulnerabilities in cold code are guaranteed to be detected). The test suite of the vulnerable OpenSSL version does not cover the Heartbeat extension. Therefore, if we run the the test suite as-is, the function that contains the Heartbleed vulnerability is never executed. PartiSan would therefore classify this function as cold and always sanitize it, which guarantees detection. To be more conservative, we executed the vulnerable function 300 times with benign input alongside the official test suite.

Next, in step 4, we compile the program with the sanitizer enabled under PartiSan. We use PartiSan's default configuration (as shown in Table 4.2) to compile each of the programs. This means that the program contains two variants of all functions, except those that are cold and those without memory accesses. We only created sanitized variants for cold functions, and unsanitized variants for functions without memory accesses. Finally, we execute our test script from step 1 a thousand times to measure the detection rate.

Out of the five vulnerabilities, ASan and UBSan detect four and three respectively. The three vulnerabilities detected by UBSan all involve an integer overflow. The overflowed value usually represents the length of some buffer, which results in out-of-bounds buffer accesses. The other two vulnerabilities are caused by a lack of bounds checking. Note that although the last CVE is classified as a heap over-read, ASan does not detect it. The reason is that the Python interpreter uses a custom memory allocator. It requests large chunks of memory from the operating system and maintains its own free lists to serve individual requests. Unfortunately, ASan treats each chunk as a single allocation and therefore is unable to detect overflows within a chunk. This shows that there is value in using multiple sanitizers that can detect different causes of vulnerabilities.

Lastly, we want to note that three out of five vulnerabilities are in code that PartiSan classifies as cold. For those cases, we manually verified that PartiSan only created the sanitized variant for the vulnerable functions. Hence, those vulnerabilities are always reported. This result supports PartiSan's underlying assumption that most bugs hide in infrequently executed code. In summary, our results show that we always detect bugs in cold code while bugs in hot code are detected probabilistically. We argue that this is a valuable property in our envisioned usage scenario: finding bugs in beta software during real usage with an acceptable performance overhead. Note that probabilistic detection is a property afforded by dynamic, but not by static partitioning.

4.6 Efficiency

We evaluated the performance of PartiSan-enabled programs using the SPEC CPU 2006 integer benchmark suite [89]. Since PartiSan clones code we also measured the size of the resulting binaries. Memory overheads—a small constant amount for the background thread and a few bytes of metadata for every function—are negligible (less than 1 %) for all

SPEC programs, so we do not report them.

We conducted all experiments on a host with an Intel Xeon E5-2660 CPU and 64 GB of RAM running 64-bit Ubuntu 14.04. We applied ASan and UBSan to all of the benchmark programs and took steps to minimize measurement noise introduced by the debugging features present in the sanitizers. For ASan we disable stack trace collection, and for UBSan we turn off error recovery, which always aborts the program instead of printing a warning message and attempting to recover for a subset of failed checks. For configurations including UBSan we also configure PartiSan to create variants of all functions, even those that do not access memory. We use the expected-cost partitioning policy with a sanitization budget of 1 %, which our runtime evenly divides across all functions. For all remaining PartiSan options we use the default values from Table 4.2.

To collect profile data we use LLVM's built-in profiling facilities on the *training* workload of SPEC. Since our chosen partitioning policy greatly benefits from profile data, we make the same data available to the baseline configuration to make the comparison fair. We compile all configurations, including the baseline, with profile-guided optimization enabled, supplying the same profile data for all configurations. When measuring the runtime, we use the *reference* workload, run each benchmark three times, and report the median.

4.6.1 Performance

Figure 4.3 and 4.4 show the run-time overheads for ASan and UBSan with respect to the baseline for all SPEC integer benchmarks. The last column depicts the geometric mean over all benchmarks, which is additionally stated in percent by Table 4.4 for easier reference.

PartiSan's partitioning without any sanitization (with two identical variants) incurs a 2 % overhead on average, with a maximum of 9 % for `gobmk`.

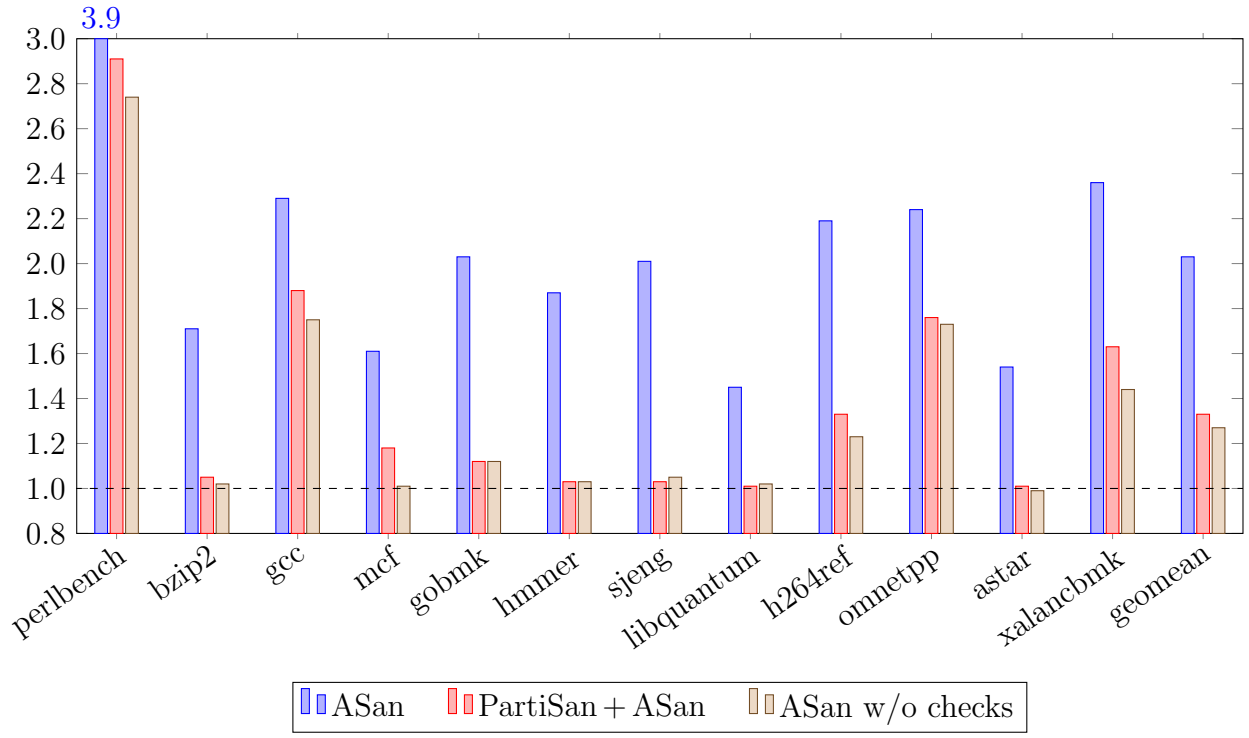

Figure 4.3: SPEC run-time overheads for ASan

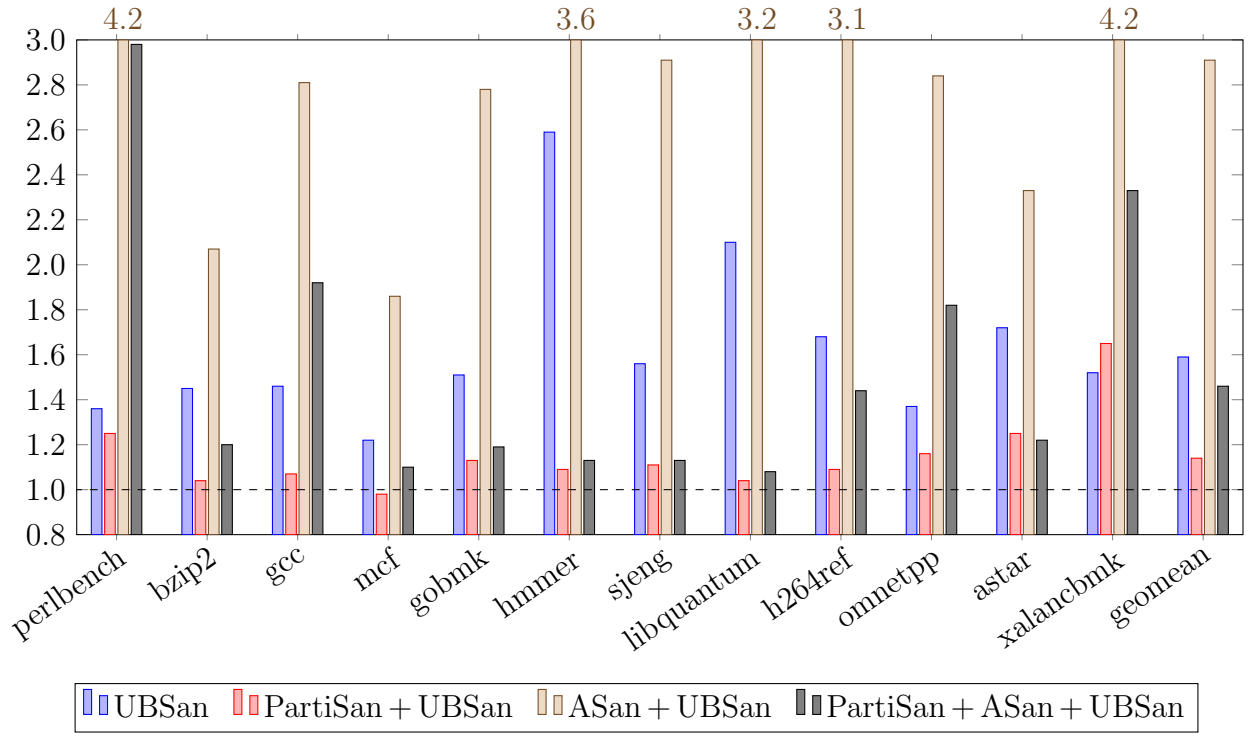

Figure 4.4: SPEC run-time overheads for UBSan and ASan+UBSan

Table 4.4: SPEC run-time overheads

Configuration	Overhead
PartiSan	2 %
ASan	103 %
ASan w/o checks	27 %
PartiSan + ASan	33 %
UBSan	59 %
PartiSan + UBSan	14 %
ASan + UBSan	191 %
PartiSan + ASan + UBSan	46 %

For the fully-sanitized versions of ASan and UBSan (absent PartiSan) we measured an average overhead of 103 % and 59 % respectively. Note that the overhead introduced by ASan can be as much as 289 % for `perlbench`.

We also created a modified version of ASan that does not execute any checks. The remaining overhead can be attributed to the maintenance of metadata and other bookkeeping tasks. This configuration represents a lower bound on the run time achievable by PartiSan since bookkeeping needs to be done in all variants. PartiSan stays close to this lower bound for many benchmarks even when using the expected-cost policy in its default configuration. For the PartiSan-enabled versions of ASan and UBSan we measured an average overhead of 33 % and 14 % respectively. This corresponds to a reduction of overhead levels by more than two thirds (68 % and 76 %) with respect to the fully-sanitized versions. We also include a configuration that enables both ASan and UBSan in Figure 4.4 to show that PartiSan can handle multiple sanitizers as long as they are compatible with each other.

4.6.2 Binary Size

Table 4.5 shows the total size of SPEC benchmark binaries in kilobytes. We state binary sizes for ASan and UBSan with and without PartiSan and the size increase in percent.

Table 4.5: PartiSan SPEC binary sizes (in kilobytes)

Benchmark	ASan	UBSan
perlbench	13,872 / 14,258 (3 %)	5,409 / 5,709 (6 %)
bzip2	10,062 / 10,147 (1 %)	2,178 / 2,231 (2 %)
gcc	20,381 / 22,314 (9 %)	12,827 / 14,396 (12 %)
mcf	9,827 / 9,836 (0 %)	1,973 / 1,981 (0 %)
gobmk	18,444 / 18,949 (3 %)	7,207 / 7,620 (6 %)
hmmer	10,844 / 10,869 (0 %)	3,001 / 3,014 (0 %)
sjeng	10,239 / 10,304 (1 %)	2,315 / 2,371 (2 %)
libquantum	9,884 / 9,914 (0 %)	2,003 / 2,023 (1 %)
h264ref	12,326 / 12,628 (2 %)	4,206 / 4,475 (6 %)
omnetpp	11,577 / 11,777 (2 %)	3,861 / 4,082 (6 %)
astar	10,039 / 10,073 (0 %)	2,166 / 2,199 (2 %)
xalancbmk	22,113 / 23,078 (4 %)	16,169 / 18,727 (16 %)
Geo. mean	(2 %)	(5 %)

The statically-linked PartiSan runtime adds a constant overhead of 6 KB to each binary. Internally, our runtime depends on the pthread library to spawn the background partitioning thread [58]. Usually, this does not increase program size as `libpthread` is a shared library.

For our SPEC configurations, the increase in relative code size is dominated by the inclusion of the ASan and/or UBSan runtime. This is due to SPEC benchmarks being relatively small programs. Therefore the larger programs in the suite exhibit the highest increase percentage wise (9 % for `gcc`/ASan, and 16 % for `xalancbmk`/UBSan). The increase in binary size over all benchmarks (geometric mean) for ASan and UBSan are 2 % and 5 % respectively. The ASan+UBSan configuration (not shown) has analogous size characteristics.

Table 4.6 gives an overview of the impact that PartiSan has on binary size for real-world programs. We state binary sizes of the programs used in our effectiveness evaluation again for both ASan and UBSan. Note that we can navigate the size versus performance trade-off by adjusting the threshold for hot code and argue that (using our policy) the maximum size increase is limited by a factor of two. In this worst case, PartiSan would classify all code as hot and create variants for all functions in the program. If binary size is a priority, the

Table 4.6: PartiSan program sizes (in kilobytes)

Program	ASan	UBSan
Php 7.0.3	20,483 / 21,983 (7 %)	8,658 / 12,536 (45 %)
OpenSSL 1.0.1f	19,128 / 25,579 (34 %)	12,153 / 14,243 (17 %)
Python 2.7.7	41,715 / 54,717 (31 %)	22,033 / 28,641 (30 %)

developer may also define a custom, size-conservative policy.

4.7 Use Case: Fuzzing

Fuzzing is an important use case for sanitization. A fuzzer repeatedly executes a program with random inputs in order to find bugs. Inputs that exercise new code paths are stored in a corpus (coverage-guided), which is used to derive further inputs (evolutionary). To aid bug detection, the program is usually compiled with sanitization. The vast majority of individual fuzzing runs do not detect bugs or increase coverage, so fuzzers rely on executing lots of runs (i.e., throughput is important). We applied PartiSan to libFuzzer [60], an in-process, coverage-guided, evolutionary fuzzing engine, with the goal of improving fuzzing efficiency.

When we first applied PartiSan to fuzzing we noticed that it represents a specific use case that benefits from a custom partitioning policy. Specifically, the fuzzer requires the program to be executed with coverage instrumentation. The gathered coverage data is similar (but not equivalent) to the profile data used for our partitioning policy. We adapted PartiSan to use online coverage data instead of profile data, which has two advantages. First, it simplifies the developer workflow since there is no need to collect profile data a priori, which would require differently-instrumented binaries. Second, it allows us to continuously refine our partitioning decisions. We integrated PartiSan with libFuzzer with minimal changes to the latter. After our changes, coverage data is consumed by two orthogonal users: libFuzzer

(to identify interesting inputs) and PartiSan (to make partitioning decisions). Additionally, the main fuzzing loop provides a natural place to make partitioning decisions. We added a call into our runtime from the fuzzing loop, forgoing the background thread in favor of synchronous partitioning.

4.7.1 Partitioning Policy

Figure 4.5 shows the main fuzzing loop extended with our partitioning policy (see Figure 2.1 for the unmodified fuzzing loop). For most functions we generate three variants: variant ① with coverage instrumentation, sanitized variant ②, and fast variant ③ without any instrumentation. At startup we activate variant ① for all functions of the program. Whenever the fuzzer discovers an input that exercises new code, we temporarily activate variant ② for the whole program and re-execute the input. Finally, if a function becomes fully-explored (i.e., all its basic blocks have been executed), we activate its variant ③.

Our policy allows us to increase coverage efficiently compared to the original program whose functions contain both coverage and sanitization instrumentation. As coverage increases, functions transition from variant ① to ③, speeding up execution of the well-explored parts of the program. The downside of this approach is that it potentially reduces the chance of bug detection as well as coverage feedback to the fuzzer. Consider an input that exposes a non-crashing bug without increasing coverage. Under our policy, such inputs execute without sanitization. Additionally, a function that we deem fully-explored might still provide useful coverage feedback to the fuzzer. The reason is that libFuzzer's coverage model is fine-grained (e.g., it includes execution counts) while our notion of fully-explored is binary. We did not find these issues to be a problem in our evaluation. The modified fuzzer discovered all bugs and never lacked a significant amount of coverage with respect to the baseline.

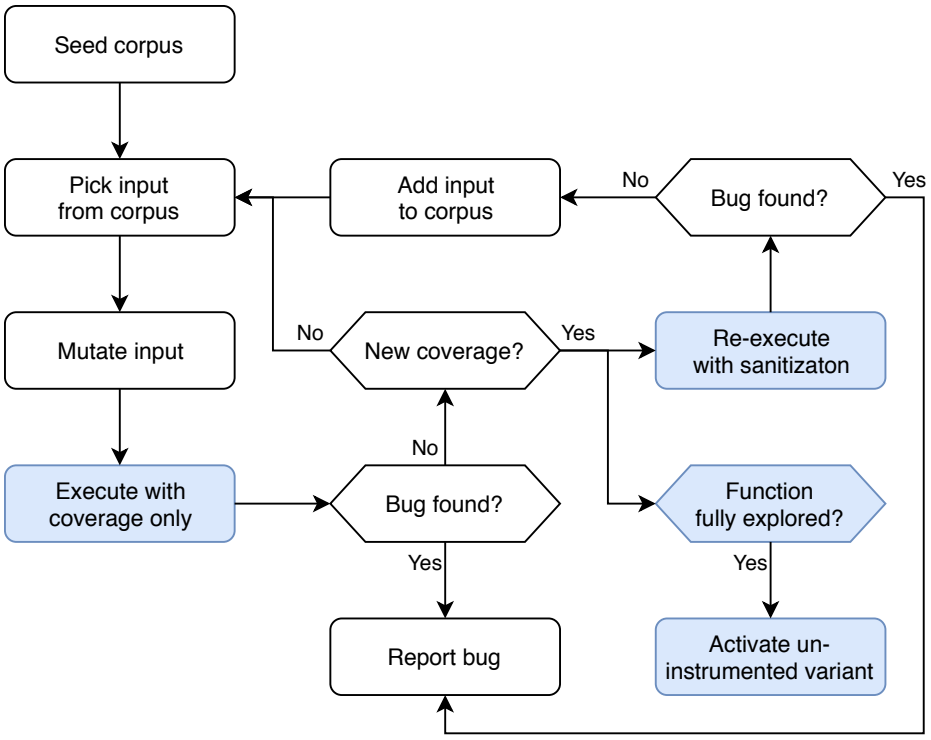

Figure 4.5: Fuzzing loop extended with PartiSan partitioning policy (in blue shade)

4.7.2 Evaluation

We evaluated the PartiSan-enabled libFuzzer on a popular benchmark suite for fuzzers [39] derived from widely-used libraries. We ran all 23 included benchmarks with ASan enabled. Out of these 23 benchmarks 11 complete (find a bug) within a few minutes. For the remaining 12 benchmarks we measured fuzzing throughput and coverage and ran them for eight hours or until completion. Figure 4.6 shows the results for two benchmarks (geometric mean of 10 runs). The markers indicate the completion of a run (i.e., after the first marker the line represents the remaining 9 runs).

As expected, PartiSan is able to increase fuzzing throughput (executions per second) for the sanitized libraries. For 9 (of 12) benchmarks this translates to improved coverage, and 3 benchmarks complete significantly faster. For example, for the `libpng` benchmark (left side of Fig. 4.6) PartiSan lets us find the bug within our time budget, whereas previously

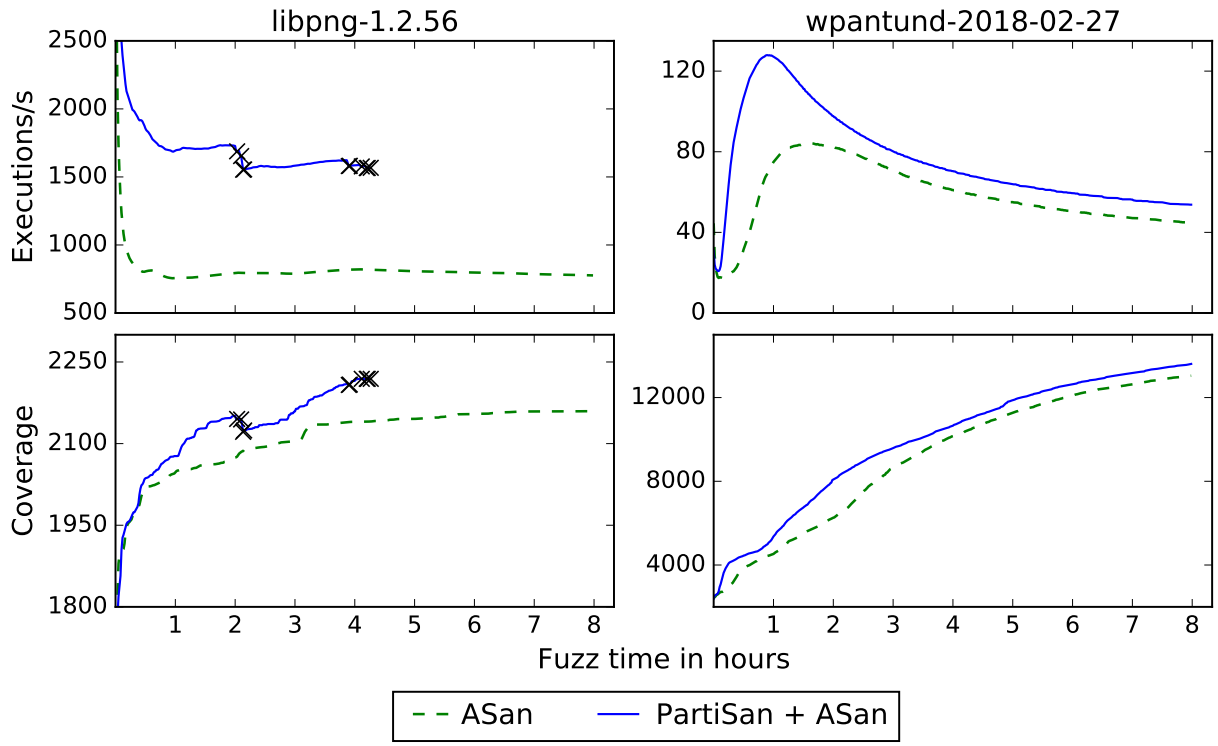

Figure 4.6: Fuzzing throughput and coverage for `libpng` and `wpantund`

we could not. However, the impact of PartiSan is not always that pronounced. For the `wpantund` benchmark (right side of Fig. 4.6), coverage only improves slightly. Note that fuzzing throughput generally decreases over time as the fuzzer explores longer and longer code paths.

4.8 Discussion

Custom Partitioning Policies We implemented three run-time partitioning policies in PartiSan. Our profile-guided and expected-cost policies can apply sanitizers at a greatly reduced cost. The flexibility of our design and implementation additionally allows developers to define their own policies. To implement a custom partitioning policy, the developer can provide her own policy function implementations when linking the final binary. The static and dynamic aspects of the partitioning policy are defined by the functions `load_policy()`

and `pick_variant()`, respectively. Our policy for the fuzzing use case is built atop this mechanism.

Asynchronous Partitioning We opted to offload our run-time partitioning procedure onto a background thread. The advantage of this approach is that, since partitioning happens asynchronously relative to the rest of the application, our runtime component has little impact on the application's performance. The disadvantage is that we cannot partition on a per-function call basis or depending on the calling context. That said, in the fuzzing use case we partition synchronously as part of the main fuzzing loop. Enabling synchronous partitioning for all of our run-time partitioning policies requires only small changes to our infrastructure.

Partitioning Granularity PartiSan partitions the program run time at function-level granularity. This design decision can have an impact on the run-time performance and responsiveness of the application. In particular, PartiSan might execute the sanitized variant of a hot function containing a long-running loop. Executing this sanitized variant can induce a noticeable slowdown as PartiSan does not support control-flow transfers between variants within the same function. Our design can be refined with finer-grained partitioning, though a significant engineering effort would be required to implement it. Our fundamental conclusions would not change with an improved partitioning scheme.

Selective Sanitization Not all sanitizers will work with PartiSan. Like ASAP, PartiSan does not support sanitizers that do not function correctly if they are applied selectively. Consider, for example, a multithreaded program compiled with ThreadSanitizer [80]. If two functions in the program concurrently write to the same memory location without acquiring a lock, then ThreadSanitizer will detect a data race. This would not be true in a PartiSan-enabled version of the program if we executed the sanitized variant of one function and the

unsanitized variant of the other. In this case, the data race would not be detected, thus rendering ThreadSanitizer ineffective.

Exploit Mitigations While we see the enablement of broader application of sanitizers as the main purpose of PartiSan, we believe it could be used to deploy exploit mitigations too. Specifically, PartiSan could lower the overhead incurred by expensive exploit mitigations by making their checks probabilistic. This could especially help to stop the unbridled replication of worms. Worms often manage to infect a great number of systems before they are detected. They can do this because their targets have unknown vulnerabilities and are only protected by exploit mitigations that have limited performance impact. PartiSan could apply stronger exploit mitigations to programs, which would enable them to detect at least a fraction of the worm attacks. This would greatly increase the chance of worms being detected early.

PartiSan Attack Surface When using PartiSan as a framework for exploit mitigation, we naturally have to consider PartiSan's impact on the attack surface of the program. In this scenario, PartiSan's variant pointer array becomes an interesting attacker target. Although this array is stored on read-only pages, we have to make it writable to run the run-time partitioning procedure. This means that there is a time window where an attacker could forcibly enable the unsanitized/unprotected function variants. PartiSan could be combined with Software Fault Isolation-based techniques to protect the pointer array from such attacks [102].

4.9 Conclusion

We have presented PartiSan, a run-time partitioning technique that increases the performance and flexibility of sanitized programs. PartiSan allows developers to ship a single

sanitizer-enabled binary *without* having to commit to either the fraction of time spent sanitizing on a given target, nor the type of sanitization employed. Specifically, PartiSan uses run-time partitioning controlled by tunable policies. We have explored three concrete policies and expect future developers to define additional, application and domain-specific ones. Our experiments show that, using our expected-cost policy, PartiSan reduces performance overheads of the two popular sanitizers, ASan and UBSan, by 68 % and 76 % respectively. We also have demonstrated how PartiSan can improve fuzzing efficiency. When integrated with libFuzzer, PartiSan consistently increases fuzzing throughput which leads to improved coverage and more bugs found.

PartiSan's dynamic partitioning mechanism enables adaptive overhead thresholds and probabilistic bug detection; neither of which are supported by static partitioning mechanisms presented in previous work. Hence, PartiSan is able to extend the usage scenarios of sanitizers to a much wider group of testers and their respective program inputs, leading to the exploration of a greater number of program paths. This will enable developers to catch more errors early, reducing the number of vulnerabilities in released software.

Chapter 5

Related Work

5.1 Code-Reuse Attacks and Defenses

The work on exploitation and exploit mitigations is extensive. Although published in 2013, Szekeres et al. [93] still gives the most complete overview of this active research area. In the following, we therefore focus on recent, closely related work on attacks and defenses. We first discuss work that relates to our novel offensive technique and then discuss defenses that relate to the one we present.

Subversive-C is inspired by the recently published counterfeit object-oriented programming (COOP) technique [22, 76] as described elsewhere in this dissertation (Section 3.2.3). COOP itself is but one of a series of exploitation techniques that are able to bypass coarse-grained control-flow integrity (CFI) policies [17, 25, 38, 77]. By virtue of exploiting dynamic dispatch mechanisms, both Subversive-C and COOP-style attacks are not stopped by randomization-based defenses such as currently used coarse-grained address space layout randomization (ASLR) techniques nor finer-grained code randomization approaches that have been widely studied in the literature [11, 21, 26, 28, 43, 44, 53]. Another related exploitation technique

is return-into-`libc` (RILC) [70, 95]. Whereas Subversive-C and COOP reuse dynamically bound methods, RILC reuses dynamically linked functions in the procedure linkage table such as those in the C standard library. Despite the name, RILC also applies to libraries other than `libc` [86].

Mobile CFI (MoCFI) [24] was designed to protect Objective-C code running on iOS for ARM[1]. MoCFI maintains a shadow stack to enforce that a return targets its original caller [23]. Further, forward indirect branches must follow a valid control-flow graph (CFG) path calculated by means of static analysis. However, Subversive-C circumvents these protections: it never violates call-return matching, and it dispatches all malicious function calls via `msgSend` which resembles a valid CFG path. Further, MoCFI's protection of the method selector and class metadata does not prevent Subversive-C since we do not corrupt selectors and avoid changing class metadata in ways that MoCFI would detect. Specifically, MoCFI ensures that the class or superclass pointer for each object is known and prevents creation of entirely new classes at run time. However, due to Objective-C's dynamic nature MoCFI must allow new classes, where only the superclass pointer is known to MoCFI. As a result, we can construct Subversive-C attacks that use valid superclass pointers or alter the method lists of existing classes. We stress that MoCFI and our novel defense complement each other and can prevent a broader range of attacks when used in concert.

CFR [72] is a compiler-based CFI implementation for Objective-C code on iOS. Unlike MoCFI, which protects returns using a shadow stack, Control-Flow Restrictor (CFR) enforces a purely static policy for all indirect branches. Since CFR does not place any particular restriction on calls dispatched via `msgSend`, CFR does not stop Subversive-C attacks but could complement our defense just like MoCFI. CFR does support programmer-inserted annotations to further constrain the CFG which could potentially prevent our attack; however, doing so requires manual effort and may lead to errors that prevent programs from running

[1]Our research uses but is not specific to x86-64 hardware.

correctly.

Readactor++ [22] is the first randomization-based defense which thwarts COOP attacks by randomizing and booby trapping C++ vtables. Due to the differences in dispatching mechanisms, the concepts behind Readactor++ does not generalize to prevent Subversive-C exploits. For example, vtables are immutable and can be hidden using execute-only memory (XoM) whereas Objective-C class metadata is mutable which is why we opted to use HMACs instead.

CPI [51] separates regular data from control data which thwarts Subversive-C exploits. CPI relies on static analysis to identify sensitive data which is more challenging for Objective-C code than C and C++ code. It also requires software-fault isolation or hardware segmentation to resist memory probing attacks [32].

Van der Veen et al. [99] presented a purely binary-based defense against COOP, which breaks data flows between vfgadgets through argument registers. Their method enforces a CFI policy derived via static code analysis that limits the vfgadget space available to an attacker, thus making attacks harder. As our example exploit relies on data flows through argument registers, it would be thwarted by this defense. However, we note that Subversive-C—and also COOP in general—does not inherently require register-based data flows, as attackers can potentially fall back to leveraging overlapping counterfeit objects only or passing data via scratch areas.

Similar to our defense, CryptoCFI [62] uses hash-based MACs (HMACs) to protect sensitive pointers. CryptoCFI computes cryptographically secure HMACs using special AES instructions on recent Intel x86-64 processors. Although a direct comparison is not possible, this defense has an overhead that is likely far higher than ours. In addition, it requires that part of the SIMD register file be reserved for CryptoCFI.

5.2 Run-Time Partitioning

Kurmus and Zippel proposed to create a split kernel with a protected partition containing a hardened variant of each kernel function, and an unprotected partition containing non-hardened variants [50]. Whenever the kernel services a system call or an interrupt request, it transfers control flow to one of the two partitions. The protected (unprotected) partition is used to service requests from untrusted (trusted) processes and devices. Unlike PartiSan, however, it does not permit control-flow transfers between the two partitions. A service request is handled in its entirety by one of the two partitions.

The ASAP framework, presented by Wagner et al., reduces sanitizer overhead by removing sanitizer checks and programmer asserts from frequently executed code, while leaving the infrequently executed code unaffected [100]. This is also a form of partitioning, as ASAP creates a sanitized and an unsanitized partition within the program. As with PartiSan, transfers between sanitized and unsanitized code are frequent with ASAP. However, contrary to PartiSan and the aforementioned work, ASAP does not create multiple variants of a function. Since partitioning happens at compile time, we consider ASAP to be a static form of partitioning. Note that static partitioning mechanisms can neither support adaptive overhead thresholds, nor probabilistic bug detection, nor our presented fuzzing policy.

Bunshin reduces sanitizer and exploit mitigation overhead by distributing security checks over multiple program variants and running them in parallel in an N-Variant execution system [103]. The key idea is to generate program variants in such a way that any specific sanitizer check appears in only one of the variants. This distribution principle makes each variant faster than the original program and also enables the simultaneous use of incompatible tools. Bunshin achieves full sanitizer coverage by running all variants in parallel, i.e., for any given sanitizer check there will be a variant that executes it. This approach improves program latency at the cost of increased resource consumption which limits Bunshin's appli-

cability. In a fuzzing scenario, for example, available cores can be more efficiently leveraged by running additional fuzzer instances.

5.3 Sanitizers

We applied PartiSan to two of the sanitizers that are part of the LLVM compiler framework, AddressSanitizer and UndefinedBehaviorSanitizer [59, 79]. Many other sanitizers exist. MemorySanitizer detects reads of uninitialized values and, although we did not include it in our evaluation, it is fully compatible with PartiSan [90].

ThreadSanitizer instruments memory accesses and atomic operations to detect data races, deadlocks, and misuses of thread synchronization primitives (e.g., pthread mutexes) in multithreaded programs [80]. Unfortunately, it is not a good fit for PartiSan because selective sanitization renders the sanitizer ineffective (see Section 4.8).

FUSS, another work by Wagner, uses a separate optimization stage to increase fuzzing throughput [101, Section 4.3]. After a warm-up phase, FUSS collects profile data from the running fuzzer. It then re-compiles the program under test using the collected profile data to omit the most costly instrumentation code, and restarts the fuzzer with the new binary. We argue that this one-time optimization through re-compilation constitutes static partitioning (albeit integrated over time), while PartiSan optimizes dynamically and continuously.

CaVer [55], TypeSan [40], and HexType [46] detect type confusion bugs in C++ programs by detecting downcasts of a base class pointer into an illegal derived class pointer. Although we did not test them, both sanitizers could potentially be applied with PartiSan.

HexVASan detects misuses of variadic arguments [12]. HexVASan detects cases where a variadic function attempts to retrieve more arguments than were passed by the caller, or

where the function casts a variadic argument into a different type than the one used at the call site. HexVASan is also a potential candidate for application with PartiSan.

5.4 Control-Flow Diversity

PartiSan partitions the run time of programs using control-flow diversity. Prior work has explored the use of control-flow diversity for security purposes.

Davi et al. presented Isomeron [26], a defensive technique to defeat just-in-time return-oriented-programming (JIT-ROP) attacks. In this context, control-flow diversity (or control-flow randomization) can be seen as the defensive evolution of code layout randomization.[2] Code layout randomization is a technique to mitigate ROP attacks, the prominent form of code-reuse attacks.

For a ROP attack an adversary strings together the execution of so-called gadgets. Gadgets are small code snippets ending in return statements, hence the name return-oriented programming. The payload of a ROP attack consists of the addresses of gadgets that the adversary wants to execute. In a classical ROP attack [82] this payload is assembled offline. Code layout randomization denies the adversary the a priori knowledge of gadget locations by rearranging the code layout in an unpredictable manner. JIT-ROP attacks overcome this defense by assembling the payload online, i.e., code pages are discovered and disassembled on the target system [87].

Isomeron defends against JIT-ROP by creating two diversified clones (containing different gadgets) for every program function and randomly switching between clones on every function call and return statement. Function returns are protected in a similar fashion using a special shadow stack to record call decisions. Since adversaries cannot predict which function

[2] Analogously, JIT-ROP can be seen as the offensive evolution of ROP.

clone the program will call or return to, they cannot craft a reliable payload. Therefore, even with precise knowledge of the gadget locations, an adversary cannot mount a reliable JIT-ROP attack, as Isomeron might transfer control flow to a non-intended location after every execution of a gadget. The defense therefore probabilistically breaks the chained execution of gadgets with the adversary's chance of success decreasing exponentially with the length of the gadget chain.

Crane et al. describe how they used control-flow diversity to mitigate cache-based side-channel attacks [20]. Side-channel attacks exploit the fact that some resources are shared across logical isolation boundaries like processes and virtual machines. In general, an adversary tries to infer sensitive information (or influence the execution) of another logical entity by observing (or inducing) leakage present in the shared resource. One such type of a leakage-prone resource are the caches placed at different levels throughout the memory hierarchy. During a cache side-channel attack adversaries measure timing differences in their own memory accesses (cache hit versus cache miss) to observe the location of memory accesses in the target program. Based on the observed memory access pattern and armed with a detailed understanding of the target program, the adversary can attempt to infer sensitive information.

Prominent cache-based side-channel attacks often target encryption routines or other cryptographic primitives. For example, attacks on specific implementations of AES and RSA are well-documented in the literature [96, 104]. In the case of AES, the implementation used a lookup table with precomputed values as an optimization. Accesses into this table are data-dependent, which allows an adversary to infer the data (which happens to be the encryption key) by observing the location (and therefore the index) of the table accesses. For RSA, the cited attack targets code accesses rather than data accesses. The attack exploits that during RSA exponentiation different code is executed depending on whether the current key bit is zero or one. In the targeted RSA implementation the difference in the leaked execution

trace is large enough to reliably infer which code path was executed. The adversary therefore learns the encryption key bit by bit.

Crane et al. create multiple variants of program functions and apply different diversifying transformations to each variant. The transformations are designed to preserve the semantics of the code, but obscure the code's memory access patterns (i.e., data access locations and execution trace). Essentially, the technique adds artificial, randomized noise to the observable leakage in the shared cache, which raises the difficulty of the attack since the adversary now has to account for this added noise.

Chapter 6

Conclusion

> *"A pacemaker is a computer that we put into our body; and a car is a computer that we put our bodies in."* – Anonymous

As computers take a bigger and bigger role in every aspect of our lives, we need to be able to depend on their correctness. A computer system that employs insecure software is only correct until the next successful attack. While we know that we should not depend on insecure software, in reality, we often do.

With our work on offensive techniques we have shown that the concept of counterfeit object-oriented programming is not limited to C++ code. Our attack, which we call Subversive-C, exploits the intricacies of the Objective-C message dispatch mechanism to perform whole-function code reuse, defeating all existing defenses. On the defensive side, we have devised a mitigation scheme against our attack technique and apply it to the Objective-C runtime. Our defense is practical since it does not require recompilation of application programs and runs complex, real-world applications with low overhead.

We also have presented PartiSan, a run-time partitioning technique that significantly decreases the performance overhead of sanitizers. With PartiSan, developers can define their

own dynamic sanitization policies such as adaptive overhead thresholds. This extends the utility of sanitizers to a wider range of usage scenarios. Building on this foundation, we have integrated PartiSan with a popular fuzzing engine. We have shown that PartiSan consistently increases fuzzing efficiency, leading to the exploration of a greater number of program paths and more bugs found.

Subversive-C and PartiSan complement each other as they tackle two different sides of the same coin. With PartiSan, developers can find more bugs before releasing software, reducing the number of vulnerabilities in shipped software. Subversive-C exposes and mitigates remaining vulnerabilities, motivating further investments into hardening existing software as well as research on the construction of bug-free programs. This is not the finish line, but one step towards more secure and dependable software.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, Alexandria, VA, 2005. ACM.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[3] 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 '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[4] S. Andersen and V. Abella. Data execution prevention. changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies. `http://support.microsoft.com/kb/875352/EN-US`, 2004.

[5] Apple Inc. AppleScript language guide. `https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html`, 2015.

[6] Apple Inc. NSXMLParser class reference. `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSXMLParser_Class`, 2015.

[7] Apple Inc. XMLPerformance on iOS. `https://developer.apple.com/library/ios/samplecode/XMLPerformance/Introduction/Intro.html`, 2015.

[8] T. M. Austin, S. E. Breach, and G. S. Sohi. *Efficient Detection of All Pointer and Array Access Errors*, volume 29. ACM, 1994.

[9] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer Berlin Heidelberg, 2006.

[11] S. Bhatkar and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[12] P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer. Venerable variadic vulnerabilities vanquished. In *26th USENIX Security Symposium*, SSYM '17, Vancouver, Canada, 2017. USENIX Association.

[13] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO*, 1999.

[14] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *Symposium on Network and Distributed System Security (NDSS)*, NDSS, 2016.

[15] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, Apr. 2017.

[16] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., 2015. USENIX Association.

[17] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[19] W. D. Clinger. Proper tail recursion and space efficiency. pages 174–185. ACM Press, 1998.

[20] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *22nd Annual Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, 2015. Internet Society.

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

[22] 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.

[23] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[24] 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 *NDSS*, 2012.

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

[26] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, 2015. Internet Society.

[27] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[28] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[29] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[30] G. J. Duck and R. H. C. Yap. Heap bounds protection with low fat pointers. In *25th International Conference on Compiler Construction*, CC '16, pages 132–142, New York, NY, USA, 2016. ACM.

[31] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

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

[33] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM.

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

[35] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[36] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[37] 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.

[38] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[39] Google. Fuzzer test suite. `https://github.com/google/fuzzer-test-suite`, 2018.

[40] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. ACM.

[41] C. Hamilton. Testing chromium: SyzyASAN, a lightweight heap error detector. `https://blog.chromium.org/2013/05/testing-chromium-syzyasan-lightweight.html`, 2013.

[42] C. Holler. Introducing the ASan nightly project. `https://blog.mozilla.org/security/2018/07/19/introducing-the-asan-nightly-project`, 2018.

[43] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[44] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

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

[46] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387. ACM, 2017.

[47] Jesse Squires. A collection of sorting algorithms implemented in Objective-C. `https://github.com/jessesquires/objc-sorts`, 2014.

[48] G. Koppen. Discontinuing the hardened Tor browser series. `https://blog.torproject.org/blog/discontinuing-hardened-tor-browser-series`, 2017.

[49] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, Jun 1994.

[50] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *21st ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1366–1377, New York, NY, USA, 2014. ACM.

[51] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Security Symposium*, 2014.

[52] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992.

[53] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, San Jose, CA, 2014. IEEE Computer Society.

[54] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2004 International Symposium on Code Generation and Optimization*, CGO '04, page 75, Palo Alto, CA, 2004. IEEE Computer Society.

[55] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium*, SSYM '15, pages 81–96, Austin, TX, 2015. USENIX Association.

[56] 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 *38th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '17, Barcelona, Spain, June 2017. ACM.

[57] E. Levy. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[58] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[59] LLVM Developers. Undefined behavior sanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2017.

[60] LLVM Developers. libFuzzer: An in-process, coverage-guided, evolutionary fuzzing engine. `https://llvm.org/docs/LibFuzzer.html`, 2018.

[61] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[62] 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.

[63] Microsoft. Best practices for security APIs: Control flow guard. `https://docs.microsoft.com/en-us/windows/desktop/secbp/control-flow-guard`, 2003.

[64] A. Milburn, H. Bos, and C. Giuffrida. SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)(San Diego, CA*, 2017.

[65] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

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

[67] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for c. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[68] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[69] National Institute of Standards and Technology. National vulnerability database. `https://nvd.nist.gov`, 2017.

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

[71] PaX. Address space layout randomization (ASLR). `https://pax.grsecurity.net/docs/aslr.txt`, 2003.

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

[73] A. Pohl, B. Cosenza, and B. Juurlink. Correlating cost with performance in LLVM. Technical report, Technical University Berlin, 2017.

[74] T. d. Raadt. OpenBSD 3.3 Release Notes. `https://www.openbsd.org/33.html`, 2003.

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

[76] 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.

[77] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[78] K. Serebryany. Sanitize, fuzz, and harden your C++ code. San Francisco, CA, 2016. USENIX Association.

[79] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference*, ATC '12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[80] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *2009 Workshop on Binary Instrumentation and Applications*, WBIA'09, pages 62–71, New York, NY, 2009. ACM.

[81] F. J. Serna. The info leak era on software exploitation. In *BlackHat USA*, 2012.

[82] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[83] 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.

[84] J. Siebert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[85] M. S. Simpson and R. K. Barua. MemSafe: Ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[86] R. Skowyra, K. Casteel, H. Okhravi, and N. Zeldovich. Systematic analysis of defenses against return-oriented programming. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.

[87] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, San Francisco, CA, 2013. IEEE.

[88] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *40th IEEE Symposium on Security and Privacy*, SP '19, San Francisco, CA, 2019. IEEE Computer Society.

[89] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark suite. `https://www.spec.org/cpu2006`, 2017.

[90] E. Stepanov and K. Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 46–55, San Francisco, CA, 2015. IEEE.

[91] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[92] R. Swiecki. Honggfuzz: A security oriented fuzzer. `https://github.com/google/honggfuzz`, 2016.

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

[94] 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*, pages 941–955, 2014.

[95] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[96] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23, 2010.

[97] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1675–1689, New York, NY, USA, 2017. ACM.

[98] V. van der Veen, L. Cavallaro, H. Bos, et al. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, pages 86–106. Springer, 2012.

[99] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[100] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, SP '15, pages 866–879, Washington, DC, USA, 2015. IEEE Computer Society.

[101] J. B. Wagner. *Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-off in Systems Software*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2017.

[102] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216, New York, NY, 1994. ACM, ACM.

[103] M. Xu, K. Lu, T. Kim, and W. Lee. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference*, ATC '17, pages 271–283. USENIX Association, 2017.

[104] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium*, SSYM '14, pages 719–732, San Diego, CA, 2014. USENIX Association.

[105] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2015.

[106] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

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