Hardware-assisted memory safety

Hans Liljestrand
Hardware-assisted memory safety

Hans Liljestrånd

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall AS1 of the school on 20 January 2020 at 12:00.

Aalto University
School of Science
Department of Computer Science
Secure Systems Group
Supervising professor
Professor N. Asokan, Aalto University, Finland

Thesis advisor
Adjunct Professor Jan-Erik Ekberg, Aalto University, Finland

Preliminary examiners
Professor Aurélien Francillon, EURECOM, France
Professor William Enck, North Carolina State University, USA

Opponent
Professor Juha Röning, University of Oulu, Finland
**Author**
Hans Liljestrand

**Name of the doctoral dissertation**
Hardware-assisted memory safety

**Publisher** School of Science

**Unit** Department of Computer Science

**Series** Aalto University publication series DOCTORAL DISSENTATIONS 8/2020

**Field of research** Information Security

**Permission for public defence granted (date)** 20 December 2019

**Abstract**
Computers today are ubiquitous and closely integrated into our everyday lives. But computers are fickle in nature. Programs are written by fallible humans and run on imperfect hardware. As a result, computer systems are plagued by memory vulnerabilities. Many remedies exist; from defensive programming techniques to memory-safe languages. But these approaches require security-expertise and costly porting of existing code. To achieve wide-spread use, we must integrate security into existing tools and languages. Moreover, this must be done with minimal performance and deployment costs. New security features are being rolled out in commodity hardware. They hold the promise of security, but are non-trivial to use effectively. In this dissertation, I show how compile-time instrumentation can use such hardware for performant memory-safety solutions.

We explore the recently introduced ARMv8.3-A PA and Intel MPX extensions. PA supports hardware-accelerated signing and verification of pointers. Not only do we address weaknesses in prior PA-based defenses, but we also present novel solutions for memory safety. In particular, we demonstrate how PA can be used for run-time type checking, precise return address protection, and stack safety. Userspace MPX-instrumentation is well-known and uses in-memory metadata to provide bounds checking of memory accesses. The kernel is paramount for security, but using MPX to protect is not straightforward. Because the kernel must manage its own memory, the MPX metadata is not feasible to use. We show how to avoid this issue using kernel-specific MPX instrumentation. But security hardware—such as the Intel SGX—can itself be vulnerable. We investigate Intel SGX side-channels, and show how compile-time instrumentation can be used to mitigate a branch-shadowing attack on SGX.

This dissertation presents security schemes that achieve minimal performance overheads by using features in off-the-shelf hardware. Our compile-time instrumentation integrates these features into existing code, without developer intervention. Together, hardware-assistance and compile-time instrumentation paves the way towards security solutions that offer optimal trade-offs in terms of development, deployment and performance costs. Yet, there are many roads ahead. Future work should explore compatibility with real code-bases, for instance, when common programming patterns rely on undefined behavior. Support should also be extended to C++ and other languages; this requires support for new language constructs such as exceptions and polymorphic types.

Memory-safe languages could also benefit from hardware-assisted security, for instance, by providing fault isolation or improving performance of existing checks. Together, these directions will allow deployable security along a broad spectrum of projects and developers.

**Keywords** Platform Security, Memory safety

**ISBN (printed)** 978-952-60-8913-3

**ISBN (pdf)** 978-952-60-8914-0

**ISSN (printed)** 1799-4934

**ISSN (pdf)** 1799-4942

**Location of publisher** Helsinki

**Location of printing** Helsinki

**Year** 2020

**Pages** 166

The road leading to this dissertation has at times been dark and gloomy. As such, I am fortunate for all the extraordinary people who have made this work not only possible but also often enjoyable. First, I cannot overstate the role my supervisor Professor N. Asokan has had on this undertaking; I do not think it would have embarked on this journey without his guidance and resolve. I have also been fortunate to collaborate with my PhD advisor Dr. Jan-Erik Ekberg and his research team at Huawei Technologies. The trust and advice afforded by Dr. Ekberg have been invaluable along this path. My research has been funded by the Intel Collaborative Research Institute for Collaborative Autonomous Resilient Systems; I am thankful for the glimpse outside academia this collaboration has afforded. I thank Professors William Enck and Aurélien Francillon for the pre-examination and evaluation of this work. I also thank Professor Juha Röning for agreeing to be my opponent in public defense of this dissertation.

I wish to thank my co-authors—Elena Reshetova, Andrew Paverd, Shohreh Hosseinzadeh, Ville Leppänen, Thomas Nyman, Kui Wang, Carlos Chinea, Zaheer Gauhar, and Lachlan Gunn—and other colleagues I have had the pleasure of working with. In particular, I wish to thank Thomas Nyman for the countless interesting discussions. I also owe a special thanks to Dr. Reshetova, who already as my MSc thesis advisor was instrumental in guiding me towards my current path. I would be remiss if I did not also thank our coordinator Dr. Niina Idänheimo who has elevated our group beyond a mere workplace. Finally, I would like to thank my family and friends whose continued support has helped me through yet another step in life.

Espoo, December 27, 2019,

Hans Liljestrand
Contents

Preface 1

Contents 3

List of Publications 5

Author's Contribution 7

Other Publications 9

List of Figures 11

List of Abbreviations 13

1. Introduction 15
   1.1 Motivation 15
   1.2 Objectives 18
   1.3 Outline and contributions 19

2. Background 21
   2.1 Compilers 22
   2.2 Memory errors, attacks, and defenses 24
      2.2.1 From stack smashing to ROP 24
      2.2.2 Control-flow integrity 27
   2.3 Memory safety 28
      2.3.1 Spatial and temporal memory safety 29
      2.3.2 Temporal memory safety 30
      2.3.3 Memory safety in C / C++ 30
   2.4 Hardware-assisted memory safety 31

3. Linux kernel memory safety 33
   3.1 Background: Intel MPX and reference counters 34
      3.1.1 Reference counters 34
      3.1.2 Intel MPX 36
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “Towards Linux Kernel Memory Safety”

I designed and implemented a GCC-plugin that enabled the use of Intel MPX within the Linux kernel. I also contributed to the design of a hardened reference counter for the Linux kernel. Together with my co-authors, I implemented over 200 patches that were subsequently accepted in the mainline Linux kernel. I wrote the paper together with my co-authors.

Publication II: “Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization”

I co-designed a control-flow randomization scheme that mitigates branch-shadowing side-channels on Intel SGX enclaves. I implemented the LLVM-based compile-time instrumentation together with my co-authors. I wrote the paper together with my co-authors.

Publication III: “PAC it up: Towards Pointer Integrity using ARM Pointer Authentication”

I co-designed a run-time type-safety scheme based on ARMv8.3-A Pointer Authentication (PA). Together with my co-authors, I investigated and proposed a mitigation to a PA-specific pointer reuse attack. Thomas Nyman and I led the writing of the paper, to which all authors contributed.

Publication IV: “Protecting the stack with PACed canaries”

I co-designed a stack-canary scheme built upon ARMv8.3-A Pointer Authentic-
Author's Contribution

tication and led the implementation of an LLVM-based research prototype. I led the writing of the paper, to which all authors contributed.

Publication V: “PACStack: an Authenticated Call Stack”

I co-designed a ARMv8.3-A Pointer Authentication scheme that provides precise return address protection. Thomas Nyman envisioned the initial design, whereas I proposed PA-specific optimizations. I implemented the LLVM-based prototype of the design. Thomas Nyman and I led the writing of the paper, to which all authors contributed.
Other Publications

The following publications are not included in this dissertation.


List of Figures

2.1 Program abstraction layers. ............................. 22
2.2 Three-part compiler construction. .................. 23
2.3 Temporal and spatial memory errors. .............. 29

3.1 Reference counters and concurrency. ............... 36
3.2 Intel MPX metadata addressing scheme. .......... 37

4.1 Intel SGX overview. .................................. 44
4.2 Branch-shadowing side-channel ..................... 46

5.1 Pointer authentication codes in pointers on ARM. 52
5.2 Validating returned addresses with a MAC chain. 54
List of Abbreviations

ABI  application binary interface
API  application programming interface
ASLR address space layout randomization
BD  bound directory
BPU branch prediction unit
BT  bound table
BTB branch target buffer
BTI branch target indicator
CCFI cryptographic CFI
CET Control-flow Enforcement Technology
CFG control-flow graph
CFI control-flow integrity
CISC complex instruction set computer
CPU central processing unit
DEP data-execution prevention
DOP data-oriented programming
DOS denial-of-Service
EC equivalence class
FP frame pointer
FSM finite state machine
List of Abbreviations

**IBRS** indirect branch restricted speculation
**IoT** Internet of Things
**IR** intermediate representation
**JIT-ROP** just-in-time return-oriented programming
**JOP** jump-oriented programming
**KASLR** kernel address space layout randomization
**LBR** last branch record
**LR** link register
**MAC** message authentication code
**MMU** memory management unit
**MPX** Memory Protection Extensions
**MTE** Memory Tagging Extension
**OS** operating system
**PA** Pointer Authentication
**PAC** pointer authentication code
**PHT** pattern history table
**PKU** Memory Protection Keys for Userspace
**PT** Processor Trace
**RISC** reduced instruction set computer
**ROP** return-oriented programming
**SGX** Software Guard Extensions
**SP** stack pointer
**SROP** sigreturn-oriented programming
**SSA** single static assignment
**TEE** trusted execution environment
**TOCTOU** time-of-check-time-of-use
**VA** virtual address
**VM** virtual machine
1. Introduction

1.1 Motivation

Computer systems are everywhere today. This has been a growing trend for decades, from personal computing and mobile phones to the Internet of Things (IoT). The transition has not been easy from a security perspective. Personal computing moved computers from physically secured server rooms into open environments. With the emergence of the Internet, computers were further opened to network attacks. In time, operating systems were forced to adapt and improve their built-in security. A similar progression happened with web technologies, which, after some initial security-woes, have become gradually more secure.

Computers have also become more integrated into our everyday lives. Even devices without obvious privacy or security concerns (e.g., toys or other gadgets) can be abused. The Mirai attacks in 2018 compromised IoT devices to mount a massive denial-of-Service (DOS) attack that crippled large parts of the Internet [7]. Security is no longer the concern of a few security-critical systems. It spans everything from smart refrigerators and gaming consoles to cyber-physical systems such as autonomous vehicles.

The widespread use of computers is not limited to the consumers; computer systems need to be designed and implemented. Easy access to software development tools and distribution channels has enabled development outside large organizations. With the recent growth of maker workshops and open hardware designs (e.g., RISC-V [12]), the same trend increasingly applies to hardware development as well. These new systems must be secured, but often the designers of these systems are not security experts. Vulnerabilities that arise from application-specific requirements and logic errors can, or must, be addressed by the developer. Memory errors are a prominent cause of many security problems. They are often unintuitive and difficult for non-experts.
Memory errors  Memory errors are faults that cause memory to be altered in unintended ways [151]. The underlying issue can be in the architecture-specific details of language implementations (e.g., how function calls are implemented). Because they depend on low-level details, they are not necessarily apparent from source code alone. As such, a higher level of security expertise and awareness is required to mitigate these attacks. The same applies to software side-channels, e.g., prime+probe attacks that exploit the central processing unit (CPU) caches [167]. Not to mention the recent transient execution attacks that leverage obscure micro-architectural behavior [27].

An attacker can exploit memory errors to alter program behavior or expose sensitive data. The stack smashing attack from 1996 [108]—and the Morris worm from 1988 [149]—exploits memory errors to inject code on the stack and then alters the function return address so that the injected code is executed on return. This attack type is prevented by widely-deployed policies such as the data-execution prevention (DEP) feature on Microsoft Windows [116]. Yet the 1997 return-to-libc attack showed that expressive attacks are possible to construct without injecting new code into program memory [130]. These attacks are mitigated by stack canaries [47], address space layout randomization (ASLR) [142], and control-flow integrity (CFI) [2], which in turn can be defeated by newer attacks, and so on.

The continuous arms race and increasingly obscure attacks require constant evolution of security mechanisms [151]. Because security today touches all areas of software development, it cannot depend on security experts or ad hoc solutions. Instead, security must be supported from the ground up, starting with the tools used to build software. Secure programming languages—such as the systems programming language Rust [114]—show promise in preventing memory errors. But new languages cannot replace the expertise and code that already exist in older languages. To protect code today, we must accommodate programming languages in current use.

Memory protection  Much research has focused on providing memory safety for memory-unsafe languages, such as C / C++. Such approaches can be categorized into compile-time and binary-only instrumentation. Binary-only approaches can be applied to compiled binaries and are convenient to use, but cannot rely on language semantics available at compile-time. Compile-time instrumentation can use language semantics such as variable scoping and type information to implement more precise security policies. Compile-time approaches can be further divided into three categories: 1) new memory-safe languages, 2) augmenting memory-unsafe languages, and 3) source-compatible hardening of existing languages.

Although memory-safe languages cannot immediately replace existing code, they are an important step towards memory safety. They are pow-
erful because security properties, such as variable ownership and array bounds, can be made first-class language constructs [114]. The same can be achieved by augmenting memory-unsafe languages such as C with new annotations and types that provide memory safety [60]. Although such hardened C variants can be applied to existing codebases, they require refactoring and code conversion, which can incur a considerable deployment cost. Finally, security mechanisms can be added to the compilation of an existing language. These approaches rely on existing semantic information and static analysis to implement security features without requiring source code changes.

**Hardware-assisted memory protection** From an instrumentation standpoint, hardening mechanisms can be purely software-based [113, 121], or rely on special-purpose hardware. Software-based solutions are convenient, but typically suffer from high performance overheads or security trade-offs (e.g., CCFI with an overhead of 50% [113], or software shadow-stacks that are vulnerable to memory disclosure [25]). Several hardware-based approaches have been proposed in the research literature (e.g., CHERI [162], HardBound [54], and HardScope [123]), but these require special-purpose hardware. Meanwhile, new security extensions are being deployed in commodity hardware (e.g., Intel Memory Protection Extensions (MPX) [125], Intel Control-flow Enforcement Technology (CET) [82], ARM8.3-A Pointer Authentication (PA) [133], ARM8.5-A Memory Tagging Extension (MTE) [9]). These features are as of yet not used to their fullest extent, or in the case of special-purpose hardware, are costly to deploy. Nonetheless, hardware-assisted memory safety provides fertile ground for building efficient security mechanisms. Features in commodity hardware are particularly interesting as they allow wide-scale deployment with no additional hardware costs.

**Kernel hardening** The operating system (OS) kernel controls the whole system and is an enticing attack target. By exploiting the kernel, an attacker can gain full access to a system, e.g., by controlling process scheduling and memory mapping. The security of the kernel is thus critical for the whole system's security. The Linux kernel is widely deployed on a range of devices, including IoT and other embedded devices. In contrast to desktop systems, it might not be possible to update embedded devices after deployment. A reactive patching strategy is thus ineffective. Kernel development must instead provide systematic approaches that eliminate whole classes of errors. Meanwhile, many memory safety efforts focus on userspace programs and cannot be directly transferred into the kernel. Intel MPX, for instance, provides configuration registers for kernel space but uses a metadata model that cannot be used within the Linux kernel [125]. But the constrained low-level environment is not always a hindrance. The homogeneous source code of the kernel can allow solutions
that are not applicable to general code, e.g., by modifying definitions in common headers and subsystems.

1.2 Objectives

The overarching objective of this work is to harden computer systems against run-time attacks that exploit memory errors or side-channels. But security hardening cannot be our only goal; solutions must also be deployable, both in terms of development cost and performance. In practice, our objective is thus to leverage existing hardware and provide source-compatible security schemes with a low-barrier of adoption. However, protecting a regular userspace application running on a compromised kernel is futile. As a consequence, the security of the OS kernel must be considered.

Compile-time instrumentation For a large class of programs, deployment costs exclude developer-intensive approaches such as switching to new or hardened languages. Compile-time instrumentation provides a solution as it uses existing source-code semantics to instrument programs with run-time checks. But instrumentation will change program behavior; and can lead to compatibility issues (e.g., when a program relies on specific undefined behavior in C / C++). In this dissertation, as a whole, I explore the research question:

RQ1 How to improve memory safety using compile-time instrumentation?

Hardware-assistance Many security mechanisms incur a large performance overhead and therefore necessitate trading off security to keep overhead within acceptable bounds. Hardware support allows for both better performance and security. But new hardware is expensive. One objective of this work is to explore the use of existing and upcoming hardware security extensions in commodity hardware. In general, I aim to answer the following research question:

RQ2 How can security features in commodity hardware be used to harden memory safety?

Kernel hardening Due to its central role, the kernel must be protected to prevent full system compromise. This work specifically targets systematic approaches that eliminate classes of memory errors, not individual bugs. Existing memory-safety research often focuses on userspace applications.
These solutions either cannot be realized in kernel space, or they require additional changes to function properly. One objective is thus exploring the adaptation of such technologies to protect the kernel. The self-contained nature of the kernel can also lend itself to more hands-on approaches that modify the kernel source itself. However, it is not straightforward to develop and apply kernel-wide patches in a systematic way. Through this work, I explore these problems, and in particular, aim to answer the following research question:

**RQ3 How can the OS kernel’s memory safety be hardened in a systematic way?**

### 1.3 Outline and contributions

This thesis is organized such that Chapter 2 offers a common background upon which subsequent chapters build. The following three chapters (Chapters 3–5) summarize the work in the five publications that comprise this dissertation. The relation between the publications and the research questions are summarized in Table 1.1. Two central themes emerge in this dissertation: compile-time instrumentation and hardware-assisted security.

Chapter 3 presents Publication I, which explores research question **RQ1**. Specifically, we address two categories of memory errors in the Linux kernel. First, we apply a kernel-wide protection scheme against reference counter overflows. We also propose an automated source code checking tool that encourages the secure implementation of new reference counters. Patches for both reference counter protection and code checking were subsequently accepted in the upstream Linux kernel [43]. Second, we also touch **RQ2** and **RQ1**, by adapting the Intel MPX instrumentation for in-kernel protection.

Chapter 4 presents Publication II, in which we address a side-channel vul-
Introduction

Vulnerability in Intel Software Guard Extensions (SGX). SGX is a userspace trusted execution environment (TEE) that provides isolated execution and encrypted memory within an *enclave*. The enclave provides both integrity and confidentiality guarantees, even against the OS. Unfortunately, SGX is vulnerable to side-channel attacks. Publication II addresses **RQ1** by employing compile-time instrumentation to prevent a branch-shadowing side-channel on Intel SGX.

Chapter 5 presents Publications III–IV, which all explore the ARMv8.3-A PA [8] hardware extension (**RQ2**). In Publication III we show how to mitigate a PA-specific *reuse attack* and implement PA-based run-time type enforcement for C. In Publication IV we show how PA can be used to harden traditional stack canaries, whereas Publication IV shows that we can eliminate the reuse attack completely for specific attacks. These works answer **RQ2** and **RQ1** by showing that compile-time instrumentation using commodity hardware can provide security with negligible performance overheads.
2. Background

Computer programs can be viewed through different layers of abstraction (Figure 2.1). A programmer does not interact directly with the hardware. Instead, they use a high-level programming language—e.g., C, C++, Java, Python—to describe intended functionality in source code. A compiler then translates the source code into machine code for a specific machine. Finally, the machine code is loaded into a physical machine that executes it. The programmer focuses on the source code, but the machine code directly affects run-time performance. Consequently, programmers often employ various techniques—e.g., inline assembly and code patterns with predictable compiler output—to produce machine code that is optimal for specific hardware. But consequences are not limited to performance; memory errors and side-channels are a direct consequence of how high-level logic is realized on specific hardware.

As an example, consider a local variable in the C language. It must be assigned a specific memory area whose size is determined based on the variable size. At run-time, the machine code does not account for variable sizes, and so might cause an overwrite memory error. Data is often stored in contiguous memory areas to improve performance. An overwrite will thus likely corrupt memory belonging to another variable, and thus be exploitable. The CPU caches memory accesses to improve performance. However, because the CPU caches are not tied to the process, they can be used as a side-channel to infer memory accesses of other processes executing on the same CPU [167].

The works included in this dissertation all use compile-time instrumentation. It is useful to consider how a compiler works in order to appreciate its relation to memory safety. Section 2.1 presents compiler internals. Section 2.2 discusses memory error exploitation techniques and defenses. Section 2.3 looks at memory safety. Finally, in chapter Section 2.4, I discuss hardware-assisted memory-safety defenses. 

\(^1\)I will not consider interpreted languages or just-in-time compilation, although most of the discussion in this dissertation also applies to them.
2.1 Compilers

A compiler typically consists of three main components (Figure 2.2): 1) a frontend that understands, parses and transforms some input language into an intermediate representation (IR), 2) an optimizer that transforms the IR to improve it without impacting the end results, and 3) a backend that transforms the IR into some destination language, typically the machine code of a targeted processor architecture. These descriptions are intentionally imprecise; for example, “improving” might mean reducing size or execution time, or minimizing memory use. The distinction between these components is also loose; in practice, they can be more or less integrated. Nonetheless, this is a useful view of the compilation process and describes the general architecture of, for instance, the LLVM / Clang compiler [106].

**Frontend** The compiler frontend recognizes a source language and transforms them into an IR used by the compiler. The frontend understands the syntax and semantics of the source language. It can thus validate the structure, and to some extent, the meaning of the program. The front-end can also perform extensive analysis by leveraging source code semantics. For instance, the Clang compiler provides various static analyzers that can be used to detect memory errors at compile time [36]. Other languages, such as Rust, perform static analysis during compilation in order to provide strong run-time security properties [114].

After validating the input, the frontend converts it to a common IR used by the compiler. In compilers such as LLVM, the IR is shared by multiple frontends for different languages. Although the IR is typically language
Figure 2.2. A compiler performs three main functions: 1) it recognizes an input language, 2) it optimizes the code, and 3) it outputs the result in some output format.

agnostic, different frontends will produce slightly different IR. The target architecture might also impose limitations that affect the produced IR. The C language, for instance, defines integer sizes that are dependent on the target architecture. Consequently, a C language frontend outputs slightly different IR on 32-bit and 64-bit architectures.

Optimizer The optimizer performs various analyses and uses the results to improve the IR. Most optimizations are performance-related. An example would be a loop optimization that moves a constant assignment out of the loop so that it is only executed once. The IR is designed to facilitate efficient optimizations and code generation. In LLVM, the IR is in single static assignment (SSA) form, i.e., each variable is assigned only once. The SSA simplifies reasoning about variable values and lifetimes.

The IR operates on an abstract machine model that has an unlimited number of registers to hold variables. Nonetheless, it does incorporate memory store and load operations, for instance, to load global variables. Optimizations then try to minimize the need for memory operations to improve performance. For instance, when a loop loads a variable from memory, it would be beneficial to move the load outside the loop. But the optimizer must not change the functionality of the program. Hence, it must first prove that the loaded memory cannot change during the loop. This might require points-to analysis that shows which pointers could point to the same memory at run-time [6, 150].

Backend The compiler backend converts the hardware-agnostic IR into machine-specific code. In the case of LLVM, this is another IR, called simply the machine IR. The machine IR syntax is shared among different architectures, which then use it to describe architecture-specific instructions. The conversion of SSA form IR must perform two critical functions: 1) instruction selection, which selects target-specific instructions that correspond to some IR instruction, and 2) register selection, which assigns each IR variable to a specific hardware register. Because the hardware registers
are limited in number, the compiler must sometimes spill variables into memory. Specifically, onto the function stack.

The stack size must be defined, for which the compiler must decide which values it needs to hold. The calling convention will typically set some constraints on the stack layout. Function parameters might be passed through the stack, for instance. The stack frame also typically holds control-data needed to implement function calls, namely: 1) the frame pointer (FP) that indicates the address of the previous stack frame, and 2) the return address, which indicates where execution should continue after the current function.

It is at this final stage, after various transformations, where memory errors materialize. Neither the IR nor machine IR know the semantics of the original input language. As such, either the original code or the various transformations after it could be based on incorrect assumptions. When these are not explicitly checked, either via compile-time verification or run-time checks, memory errors occur. Moreover, hardware limitations mean that control-data such as the return address is stored interleaved with other data on the stack. This allows seemingly small memory errors to affect program operation drastically.

### 2.2 Memory errors, attacks, and defenses

It is useful to review the history of memory attacks and defenses in order to appreciate the current state of run-time memory protection. In this section, I will present the early progression of attacks, from code injection [149, 108] to return-oriented programming (ROP) [141]. Along the way, I will present the evolution of related defenses and their subsequent circumvention techniques. I will finally present current research and directions in memory protection.

#### 2.2.1 From stack smashing to ROP

**Code injection attacks** Buffers allocated on the stack have a specific size at run-time. A stack-based buffer overflow happens when a write to such a buffer exceeds that size and corrupts other memory. An overflow could happen because the size of input provided by an attacker $\mathcal{A}$ is not verified to be within expected size limits. The traditional *stack smashing* attack exploits such overflows to inject attacker-controlled machine code onto the stack [149, 108]. To execute the code, $\mathcal{A}$ alters the program control flow by exploiting the implementation of functions.

On complex instruction set computer (CISC) architectures such as x86, a function call implicitly stores the return address on the stack before transferring control into the function. Correspondingly, the return instruc-
Background

tion then implicitly loads the return address from the stack and transfers control to the pointed-to address. On reduced instruction set computer (RISC) architectures, e.g., ARM, the return address is stored in the link register (LR) on function entry [10]. Nonetheless, it must then be stored on the stack to facilitate nested function calls that overwrite LR. In both cases, the return address is thus stored on the stack. A can thus not only inject the machine code but also corrupt the return address such that it points to the injected code.

**W X policies** W X policies enforce that memory is either writeable or executable but not both at the same time. They effectively prevent code injection attacks by making the stack non-executable. A non-executable stack does not stop memory errors from corrupting memory. Instead, it prevents A from executing the injected code. Today, W X policies are widely supported by operating systems, including Microsoft DEP [116] and Linux [129].

**return-into-libc** The 1997 return-into-libc attack demonstrated that injected code is not necessarily needed [130]. In this attack, A replaces the function return address with the address of a libc function. The victim will then incorrectly return to the injected address and execute the attacker-chosen libc function. This technique allows A to effectively call arbitrary functions. Moreover, because the stack grows towards lower addresses, A can corrupt multiple stack-frames. A can thus inject several return addresses such that the program executes multiple functions of the attackers choosing.

**Stack canaries** Although the return-into-libc attack is sophisticated, it still exploits a stack buffer overflow. As such, it can be prevented by detecting overflows before the function return is executed. Stack canaries, proposed initially in StackGuard [47], build upon this realization. A stack canary is a value that is stored on the stack such that it will be corrupted if an overwrite reaches the return address. The integrity of the canary can then be verified before the function returns.

A can circumvent canary defenses by guessing the correct canary value, or by corrupting non-protected data on the stack [1, 135]. Overflows that exploit string functions can be prevented using terminator canaries that include string-terminating characters [46]. Early defenses prevented canary copy and re-use attacks by masking the canary with the return address [61]. Several hardened canary schemes exist, including DynaGuard [131] and DCR [76] that allow re-randomizing canary values at run-time; and polymorphic canaries [160] that allow efficient diversification of canaries in forked processes.

In practice, canary schemes are very similar to the original StackGuard. Both GCC and Clang provide stack protection using canaries [70, 35]. The instrumentation initializes a single process-wide canary at startup.
The compilers offer different variants of the stack protector, but these only control which functions are instrumented. For instance, a canary in a function without buffers is unlikely to be useful, and so it can be omitted. Ultimately, any canary scheme can be circumvented by avoiding overwriting the canary or by leaking its value. This inherent weakness makes the performance cost of hardened canary schemes questionable.

**Code-reuse attacks**

W X policies or canaries can not prevent all return-into-libc attacks, or, more generally, code-reuse attacks. ROP is an advanced code-reuse technique that can be used to realize expressive attacks [141]. In ROP, \( \mathcal{A} \) changes multiple return addressees on the stack such that they point into gadgets. In contrast to return-into-libc, a gadget is not necessarily a complete function. Instead, it can be any section of code that ends in a return. The return is used to chain gadgets together. \( \mathcal{A} \) can use ROP for arbitrary computation by finding and using different gadgets, e.g., memory load and store gadgets.

Subsequent research has demonstrated that ROP attacks are possible on ARM architectures [97, 49]. Others have demonstrated that jump-oriented programming (JOP) attacks can achieve similar expressibility without return instructions [31, 18]. Instead, JOP attacks use sequences of instructions with similar behavior. For instance, instead of a return, a JOP gadget could end with an indirect branch instruction. Other attacks exploit the behavior of abnormal control-flow transitions, such as signal handlers. When entering a signal handler, the execution state is stored in a signal frame on the program stack. On return from the handler, the sigreturn system call restores the prior execution from the signal-frame and resumes execution. Sigreturn-oriented programming (SROP) attacks exploit this behavior by writing a bogus signal-frame on the stack and performing a sigreturn system call [19].

**Probabilistic defenses**

Code-reuse attacks depend on finding gadgets. A typical program contains a large amount of code, and so gadgets are typically available. \( \mathcal{A} \) can thus analyze a target binary, identify the gadgets, and store their memory addresses. Address space layout randomization (ASLR) randomizes the location of different memory regions to remove predictable gadget addresses [128, 163]. To perform a code-reuse attack, \( \mathcal{A} \) must either leak addresses or attempt to guess them, which likely causes the program to misbehave or crash. ASLR can also be applied to protect the OS kernel [92]. Both Windows [87] and Linux [42] now make use of ASLR to protect both the kernel and userspace applications.

Side-channel attacks can break ASLR in both userspace [73, 62] and in the kernel [80]. The kernel is at a disadvantage because it is long-lived and interacts with multiple different programs. Any program, log, or driver could be used to leak kernel addresses for an attack launched from another process. It is likely also non-trivial to transfer ASLR hard-
ening schemes, such as re-randomization [112, 161], to a kernel setting. Moreover, advanced attack techniques such as just-in-time return-oriented programming (JIT-ROP) can discover new gadgets while executing a ROP attack [145]. JIT-ROP works around fine-grained ASLR schemes, but would likely apply equally to re-randomization schemes.

2.2.2 Control-flow integrity

**Stateless CFI** Defenses such as w, x policies efficiently prevent code-injections attacks. Defenses like ASLR and canaries make code-reuse attacks harder, but do not prevent them entirely. CFI directly addresses code-reuse attacks by enforcing a policy based on a pre-computed control-flow graph (CFG) for the protected program [2, 3]. Instrumentation then checks that the forward-edges, e.g., indirect function calls, target a destination containing an expected identifier. The identifier is based on an equivalence class (EC) derived from the CFG, such as that functions that share call-sites belong to the same EC. This can result in an overly permissive policy [3]. Another approach is to route all indirect function calls via jump-tables [153]. This allows the implementation of more restrictive policies. Nonetheless, carefully constructed code-reuse attacks can circumvent stateless CFI solutions [137, 51, 72, 40].

**Fully-precise CFI** Fully-precise CFI is an idealized stateless CFI policy. It only allows control-flows that happen in some benign execution of a program. Unfortunately, if the benign execution allows multiple functions at a call-site, then even a fully-precise CFI must allow those targets. A can thus still alter control-flow among those possible targets. Subsequently, even fully-precise CFI has been shown vulnerable to control-flow bending attacks [29].

**Shadow call stacks** Backward-edges in the CFG—for instance, a function return—can be enforced using a shadow call stack [25]. The shadow call stack is used to hold a secure copy of return addresses, and thus allows precise verification of returns. The shadow call stack must be protected for it to guarantee protection. It can be protected using software fault isolation [2]. Software-based shadow call stacks can be faster by placing them in the same address space and addressing them via a dedicated register [48, 25]. However, such approaches can leave the shadow call stack open to attack.

**Context-sensitive CFI** Shadow stacks are stateful, i.e., they do not follow a fixed statically determined CFG. However, they only protect function returns. Recent work has proposed solutions that combine static analysis and run-time tracking to implement stateful CFI policies. Such approaches restrict control-flows with respect to the whole execution path of the program [56, 111, 68]. The program execution flow is tracked using the
Intel Processor Trace (PT) [83, Vol.3C, Chap.35] and monitored from a separate monitoring process. This approach has been demonstrated to allow enforcement of unique control-flow targets [78]. Unfortunately, this does not come for free; the performance overhead of instrumentation is less than 5%, but the monitoring process incurs an additional overhead of about 10% [78].

**Protecting code-pointers** Control-flow attacks generally depend on manipulating code pointers (including the return address). A natural alternative to CFI is thus to focus on the protection of code pointers. *Code-pointer integrity* takes this approach [101]. It uses static analysis to identify and protect pointers by placing them in protected memory, called the *safe stack* [102]. An alternative to isolating pointers, is to cryptographically protect them. Cryptographic CFI (CCFI) uses message authentication codes (MACs) to verify the integrity of pointers before they are used [113]. The MAC is derived from the pointer’s address and its type. Using a type-based MAC permits efficient instrumentation without the need for extensive static analysis. Approaches that protect code pointers provide exact CFI verification.

**Clang CFI** CFI implementations have been demonstrated on both GCC and LLVM compilers [153]. The LLVM / Clang compiler provides a stateless CFI mechanism [37]. In contrast to traditional CFI approaches, Clang uses type-based checking at call sites. In effect, it checks that indirect function calls are performed using pointers of the correct dynamic-type. In addition, Clang supports software-based shadow call stacks on 64-bit ARM architectures [38].

**Data-only attacks** Performance and compatibility questions aside, current defenses proposed in the literature offer comprehensive protection against code-reuse attacks. However, non-control data attacks do not alter any control-data, and therefore, do not violate CFI policies. Such attacks have been demonstrated to be possible on real-world applications [33]. Moreover, recent data-oriented programming (DOP) attacks show that non-control attacks can be used to perform arbitrary computations [79]. DOP is similar to ROP, but it does not directly alter control-flows to execute gadgets. Instead, ∆ alters data that affects control-flow decisions—e.g., loop iterations—to chain and select gadgets.

### 2.3 Memory safety

The attacks described in Section 2.2 are a result of memory errors and would all be prevented by guaranteed memory safety. Despite such widespread concern, the definition of memory errors remains elusive. There are efforts to formalize the notion of memory safety. This is valuable because it would
allow the memory safety of a specific program to be verified. Unfortunately, the memory safety of a C program has been shown to be undecidable in the general case [136]. Other models have shown undecidability under multi-threading [169]. Nonetheless, recent work has moved towards more practical definitions that could possibly be applied to subsets of C / C++ [14].

In practice, memory safety is often defined as the absence of memory errors. A memory error can be loosely defined as an error that causes unintended changes to memory. In the literature, the memory-error definition often depends on the research goals [77]. A common definition of memory safety is the absence of specific memory errors. For instance, Younan et al. [168] lists the following memory errors: 1) stack-based buffer overflows, 2) heap-based buffer overflows, 3) dangling pointer references, 4) format string vulnerabilities, and 5) integer errors. Even without an exact definition, it is intuitively clear that memory safety would prevent a large class of attacks.

### 2.3.1 Spatial and temporal memory safety

Szekeres, Payer, and Wei [151] categorize errors into spatial and temporal errors. When a memory read or write accesses memory out of bounds, a spatial memory error occurs. For instance, if an array is accessed with an index beyond its allocated size. A temporal memory error occurs when data is accessed after its allocated memory has been freed. For instance, use-after-free errors that occur when a variable is used after its allocated memory has been freed for other use. Although this separation does not provide a formal model, it highlights differences in both attacks and defenses. Figure 2.3 illustrates the difference between temporal and spatial memory errors.
Spatial memory safety

Spatial memory safety is conceptually clear: a memory read or write should not touch unintended memory. In statically-typed and type-checked languages, unintended memory is often unambiguously specified by the type. The type then allows easy compile-time verification and run-time checking of memory accesses. C / C++ are statically-typed but allow casting to and from raw pointers. Raw pointers do not carry type (or size) information, and so do not support run-time type checking. A spatial memory error can corrupt the memory of unrelated data. As seen in Section 2.2, such errors can be used to implement expressive attacks.

Temporal memory safety

Temporal memory safety is, again, conceptually simple: a variable should not be used after its memory is deallocated. Some languages employ techniques such as garbage collection to deallocate memory automatically. Rust uses the concept of variable ownership and automatically deallocates memory when an owned variable goes out of scope [114]. In the context of C / C++, memory can be manually allocated and deallocated. A program must exercise care to deallocate an object only when it is no longer used. The use of shared objects can be tracked using reference counters in order to ensure safe deallocation [39]. When an object is prematurely deallocated and used afterward, a use-after-free error occurs. Because the memory might already have been assigned to other data, such errors are exploitable [168].

Memory safety in C / C++

There are many proposals for reaching memory safety in the C language. These can be roughly split into source-compatible approaches that do not require source code changes, and those that do. The latter consists of approaches such as Cyclone [86] and CCured [122], with CCured having a reported overhead of 25–214% in SPECINT 95 [122]. Checked-C takes the middle-ground by providing new checked pointers that can be used to convert C codebases gradually [60]. Nonetheless, even partial conversion incurs relatively high-overhead, with a reported average of 8.6%.

Source-compatible approaches use existing source code semantics and static analysis to implement memory safety checks. One approach for spatial memory safety is to protect allocation bounds, e.g., Purity [75] and AddressSanitizer [139]. These approaches use a shadow memory to define memory red zones that can be used to detect spatial and temporal memory errors. But they still incur a high overhead: AddressSanitizer has a reported slowdown of 73% [139]. AddressSanitizer is currently widely used in testing [11] and supported in the mainline Clang compiler [34]. However, allocation-based checking cannot detect inter-object overflows and has been shown vulnerable to attacks [69].
Another approach is to tie bounds to the pointer. Pointer-based bounds have the advantage that they can be narrowed. Narrowing can be used to limit a pointer to sub-structure within an object, instead of the allocation that encompasses the whole object. Pointer-based bounds can be realized with so-called fat pointers, i.e., a modifier pointer representation that includes its bounds [89]. Fat pointers are convenient, but cause compatibility errors due to the changed pointer representation. An alternative is to use disjoint metadata to track pointer bounds [127, 164, 54]. Baggy bounds checking is one such approach and incurs an overhead of 72% on SEPECINT 2000 [5]. SoftBound uses a similar approach but supports write-only checking to achieve an average overhead of only 15% [121].

2.4 Hardware-assisted memory safety

Due to the high overhead of most run-time protection schemes, there is a vested interest in providing hardware support. This includes hardware-assisted CFI (e.g., HAFIX [50] and CFI CaRE [124]) and run-time scope enforcement [50]. CHERI is a MIPS-based architecture that provides memory safety by using capability-based memory addressing [162]. CHERI capabilities have been shown to support Unix-like kernels and full software stacks [52]. Such research is valuable, but faces deployment challenges due to the need for hardware changes. However, hardware manufacturers have recently started rolling out new memory-safety primitives in their processors.

Intel Memory Protection Extensions (MPX) is a hardware-assisted spatial memory safety feature introduced in the Skylake CPUs [125]. It supports pointer-based bounds checking similar to SoftBound [121] by providing new instructions for accessing bound metadata and checking bounds. Intel MPX is further explored in Publication I (Chapter 3). Intel Memory Protection Keys for Userspace (PKU) is an extension that can enforce that a protected memory region is accessed only when the process has enabled a specific key. In practice, the process itself tags memory regions for use with a specific key, and then loads that key into a configuration register. ERIM uses PKU to implement in-process isolation for userspace programs [155]. An upcoming feature in Intel processors is Control-flow Enforcement Technology (CET) [82]. It includes two components, a hardware-assisted shadow stack—similar to the software shadow call stacks discussed in Section 2.2.2—and indirect branch tracking. Indirect branch tracking enforces that the execution of a call or jump instruction is always followed by the execution of a special instruction that marks a valid target.
ARM Pointer Authentication (PA) is a new feature introduced in the ARMv8.3-A architecture [8]. It provides cryptographic primitives to sign and verify pointers. It is used in Publications III–V and is further discussed in Chapter 5. New features in the later ARMv8.5-A architecture include branch target indicator (BTI) and Memory Tagging Extension (MTE) [8]. BTI is similar to the indirect branch tracking in Intel CET, although it allows separation of indirect function calls and indirect branches. MTE is somewhat similar to Intel PKU, but ties access permissions to pointers, not the process. A hardware-assisted variant of the AddressSanitizer has been demonstrated using MTE [140].
3. Linux kernel memory safety

The OS kernel is a security-critical piece of software that controls userspace applications. The Linux kernel is a unikernel, i.e., a monolithic kernel that controls everything from process scheduling to storage devices and networking. From an attacker’s perspective, it is a lucrative target. Unfortunately, the Linux kernel is also written in C, which readily lends itself to memory errors (Section 2.2). Protecting the kernel is thus important. It is tempting to consider the application of userspace memory-protection schemes to the Linux kernel. But this poses three challenges:

C1 The kernel uses programming patterns that are incompatible with many established userspace defenses.

C2 The adversary model is different, i.e., the attacker \( A \) operates outside the kernel.

C3 Many security mechanisms depend on kernel facilities or use it to provide integrity and confidentiality guarantees.

C1 Incompatibilities depend on the security policy and its implementation. For instance, pointer-based memory safety could benefit from narrowing (Section 2.3.3). But the kernel implements inheritance using a model that is incompatible with narrowing. A subclass in the kernel is defined by placing the base class as a substructure at the end of the inheriting structure. If a pointer to the base class were narrowed, it could no longer access the inheriting class. Conflicts between the security policy and kernel programming patterns can be solved either by weakening the security policy or by modifying existing coding patterns.

C2 In users-space adversary models for memory safety, \( A \) typically attacks a process from within it. \( A \) aims to alter the behavior of the same process they interact with (Section 2.2). The kernel has a fundamentally different adversary model. It is attacked from outside and cannot be trivially reset. In userspace, it is relatively cheap to reset a misbehaving process without affecting the system as a whole. This allows defenses that
kill the process outright. This is not the case for the kernel; resetting the kernel affects all running applications and could in itself be considered a DOS attack. In the kernel adversary model, $\mathcal{A}$ can typically launch arbitrary userspace programs, which then interact with the kernel, e.g., through system calls. The kernel can terminate an offending process, but this neither resets kernel defenses nor prohibits $\mathcal{A}$ from retrying with a new process. This is particularly problematic for security mechanisms that rely on probabilistic defenses, e.g., kernel address space layout randomization (KASLR) [42].

C3 Userspace defenses often rely on services by a higher privilege level, e.g., the kernel. In the case of Intel MPX, the mechanism relies on the kernel to dynamically manage memory mappings in order to improve performance. But the dependency could also affect security, for instance, if the kernel is used to protect sensitive data. While the kernel could rely on the hypervisor, this might cause unreasonable performance overhead and complicate the implementation of such defenses.

Practical kernel protection must consider both the specific run-time environment and the different adversary model. But the monolithic code base of the Linux kernel also has some advantages. It follows strict guidelines and coding practices, which lends itself to systematic protections. The kernel also uses its own Makefile-based build system kbuild [28], which allows clean integration of security features. Linux has also introduced kbuild support for GCC-plugins [44], and more recently added support for building the kernel with LLVM / Clang [105]. In Publication I, we leverage these aspects to address two classes of bugs within the Linux kernel: temporal memory errors caused by reference counter overflows, and spatial memory errors.

3.1 Background: Intel MPX and reference counters

3.1.1 Reference counters

As mentioned in Section 2.3.2, reference counters are a technique for the safe deallocation of objects [39]. A reference counter is conceptually simple: it is an integer that tracks the number of references to an object [39]. When the count reaches zero, this implies that the referenced object is no longer used and that its memory can be freed for other uses. An ideal reference counter provides only an initialization instruction, and conditional increment and decrement instructions. The exact value of the counter is not needed, and it is sufficient for the increment and decrement to return success on non-zero values.
Reference counter overflows Integer overflow is a memory error [168] that results in undefined behavior [85, 55]. On most contemporary processor architectures, an overflow will follow two’s complement semantics, i.e., an integer with the value \texttt{INT\_MAX} will be set to \texttt{INT\_MIN} when incremented. Integers in Linux kernel always follow this behavior due to compiler configuration.\(^1\) A reference counter can thus overflow and incorrectly reach zero through repeated incrementing.

The number of references is typically bound by resource limitations. Hence, a reference counter remains small enough to avoid integer overflows. However, a program counter error could allow the counter to be incremented without allocating resources. This could allow an attacker to increment the counter indefinitely, thereby causing the counter to overflow. This will trigger object deallocation and subsequently cause a use-after-free error (Section 2.3.2).

In the context of the Linux kernel, reference counter overflows have been shown to be exploitable.\(^2\)

Concurrency and reference counters Reference counter semantics must remain sound when accessed concurrently. The counter must be atomically and conditionally modified on object acquisition and release. This prevents time-of-check-time-of-use (TOCTOU) type errors that could result in use-after-free errors (Figure 3.1). Modern CPUs support out-of-order execution that allows the CPU to reorder instructions on a micro-architectural level as long as the architectural result remains the same. For example, the CPU might reorder load instructions to improve the locality of memory loads. Memory barriers, i.e., instructions that guarantee partial or full ordering of memory reads and writes, can be used to prevent this.

Reference counters in the Linux kernel To avoid concurrency issues, the Linux kernel versions prior to v4.14 implemented reference counters using the \texttt{atomic\_t} type. It provides an atomic application programming interface (API) for an integer and guarantees partial memory ordering [115]. Memory ordering is guaranteed either using architecture-specific memory barriers, or a generic implementation based on spin-locks. The kernel also provides the \texttt{kref} type, which is designed for reference counting and implemented with \texttt{atomic\_t} [99, 115]. Unfortunately, \texttt{kref} is sparsely used, leading to a multitude of hand-crafted reference counter implementations based on \texttt{atomic\_t}.

Reference counters in the Linux kernel often serve multiple purposes. For instance, the \texttt{sk\_wmem\_alloc} reference counter also tracks the message transfer queue for the network layer. Hence, it needs to read the exact counter value and perform arbitrary additions or subtractions. To further

\(^1\)This behavior is guaranteed irrespective to CPU architecture by using the -fwrapv and -fno-strict-overflow compiler flags.

\(^2\)Reference counter errors were exploited, for instance, in CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2017-7487 and CVE-2017-8925.
complicate matters, `atomic_t` is not exclusive to reference counters. Due to such varied use, reference counters cannot be automatically identified and found without developer intervention.

### 3.1.2 Intel MPX

The Intel Memory Protection Extensions (MPX) is a pointer-based spatial memory safety mechanism (Section 2.3.1). It debuted in the Skylake architecture [134, 125]. MPX provides new instructions for managing and checking pointer bounds; and new registers for configuring MPX and storing bounds. The pointer bounds are stored in separate metadata without changing the representation of pointers. The Intel ICC compiler supports MPX, and GCC added support in GCC 5.0 [125], although the latter has since version 9.1 dropped support [67]. Because MPX is pointer-based, it can perform narrowing (Section 2.3.3) and could potentially avoid vulnerabilities inherent to allocation-based bounds checking [69]. However, in practice, the GCC instrumentation must make concessions to avoid compatibility issues from strict narrowing [125].

**MPX and temporal safety** MPX is not designed to provide temporal memory safety, nor does it do so. The pointer used to free dynamically-allocated memory could be invalidated by setting its bounds to zero; however, there is no mechanism by which to invalidate other pointers to the same memory. Hence this would not prevent use-after-free errors. Oleksenko et al. [125] propose adding a lock-and-key mechanism that provides temporal safety using current MPX hardware.
Figure 3.2. MPX bounds metadata uses a two-level mapping based on a pointer’s address.

**Pointer bounds data** When accessing a pointer to stack-based data, the compiler can statically inject bounds information without the need for additional metadata. This is not always possible, e.g., when loading a pointer from shared global memory. For such cases, MPX provides the `bndstx` and `bndldx` instructions that store bounds in disjoint metadata. To check the bounds, they are first either loaded from the metadata or statically set by the instrumentation. The check is then done with the `bndcl` and `bndcu` instructions, which check the lower and upper bounds, respectively. Notably, a bounds check is performed in three distinct operations: 1) one to load or set the bounds into a bounds register, 2) one to check the lower bound, and 3) one to check the upper bound. This can cause TOCTOU type errors in multi-threaded applications because the pointer and its bounds are loaded as separate non-atomic actions.

**Accessing pointer metadata** The MPX bounds metadata is indexed based on the pointer’s address using a two-level mapping, via a bound directory (BD) to a bound table (BT) that contains the bounds (Figure 3.2). The metadata must be managed by the software; the hardware instructions only accelerate the addressing and lookup. On GNU/Linux, the program reserves the address space for the BD and writes its address into a configuration register. On 64-bit systems, this is a 2GB memory range. Each 64-bit entry in the BD points to a 4MB BT, which in turn contains bounds for the pointers. The memory pages of the BD are mapped to physical memory only when written to. The individual BTs are similarly mapped on-demand; when `bndldx` / `bndstx` encounters an empty BT entry, they trigger a fault. The kernel manages the fault by mapping the 4MB memory space for the BT and then allows the userspace process to continue transparently.
3.2 Results: refcount_t and MPX in the Linux kernel

3.2.1 Preventing reference counter overflows in Linux

In Publication I, we present our work on protecting reference counters. We answer three questions: 1) how to properly handle reference counter overflows, 2) how to design an API for the Linux kernel, and 3) how to apply kernel-wide changes efficiently?

Handling of reference counter overflows Reference counter overflows must not be exploitable by causing the counter to reach zero while the referenced object is still in use. Performance requirements prevent heavy-handed approaches that maintain the value, e.g., by dynamically switching to a larger integer type [55]. But the vulnerability can be mitigated by saturating a reference counter on overflow [20]. A saturated counter is locked at its maximum value. Because the correct value is lost, the counter cannot be safely incremented or decremented after saturation. Instead, a saturated counter always returns success when incremented or decremented without changing its value. In effect, this converts an exploitable use-after-free error to a wasteful but otherwise harmless memory leak.

Design of refcount_t We began our work by analyzing the design of the PAX_REFCOUNT feature of the grsecurity Linux patches [20]. It first introduced the saturation mechanism by adding it directly to the atomic_t type. PAX_REFCOUNT also introduces a new atomic_unchecked_t type to support uses that depend on integer overflow. This unintuitively changes to atomic_t behavior and requires changes in unrelated code that relied on the old behavior.

We initially explored the possibility of porting PAX_REFCOUNT to the mainline kernel. However, after initial porting efforts, a new refcount_t type was proposed by Peter Zijlstra [171]. It uses the saturation mechanism described above and also logs overflows to facilitate the fixing of the bugs that caused the overflow. Because refcount_t is used for reference counting only, its semantics are self-documenting. Moreover, it incorporates compile-time checks that discourage unsafe use (e.g., ignoring return values of refcount_t functions). Based on our porting efforts and analysis, we proposed API improvements that allowed for wider adoption by accommodating existing use cases.

Converting to refcount_t To support future development and aid in our kernel-wide conversion efforts, we developed Coccinelle [152] patterns for reference counters. Coccinelle is a text matching and transformation tool that is used for automated patching and detection of problematic code. While it could not unambiguously distinguish reference counters from all other atomic_t uses, it proved useful in detecting candidates for refcount_t.
conversion. Not only did we use our Coccinelle patterns to create over 200 accepted patches, but we also integrated the pattern with the kernel's Coccinelle code-quality checks.

### 3.2.2 Using Intel MPX in the Linux kernel

MPX ostensibly supports kernel protection; for instance, by including configuration registers for ring-0 (i.e., kernel-space) execution. However, the metadata scheme cannot be trivially realized within the kernel. Not only because of the high overheads observed in userspace applications but more fundamentally due to the reliance on on-demand mapping. The kernel, as-is, cannot handle page-faults caused by itself. This means that the whole BD must be mapped to physical memory, or somehow always pre-emptively mapped before use. The BT allocation faces the same issue. Even an optimized approach that only reserves metadata for used kernel memory would increase memory use by 500% (e.g., for each possible 64-bit pointer a 64-bit BD-entry, and a 256-bit BT-entry).

**Avoiding bounds metadata** Our work in Publication I approaches this challenge by first realizing that the kernel already tracks its own memory allocations. We devised a way to utilize the kernel memory allocator to retrieve allocation-based bounds instead of using `bndstx / bndlndx`. In practice, our instrumentation removes any bound metadata stores and replaces loads with a function that retrieves the allocation bounds from the allocator. This change removes additional memory requirements imposed by MPX. However, it also changes the pointer-based bounds checking to a mixed model that, in some cases, uses allocation-based bounds.

**Implementation of MPXK** Our prototype implementation, MPXK, builds on the prior GCC MPX instrumentation. MPXK adds new runtime functions to the kernel for loading allocation bounds. It then applies GCC MPX instrumentation. To apply our modifications, we used the new GCC compiler-plugin infrastructure and implemented our own MPXK plugin [44]. Our plugin replaces any metadata loads with calls to our added in-kernel functions.

### 3.2.3 Discussion: tricky bugs and environments

**Detection reference counter bugs** Reference counter errors can be subtle and seldom cause directly observable side effects. In practice, even faulty reference counters are unlikely to overflow under benign conditions. The vulnerable code path would need to be exercised $2^{\text{integer\_bit\_size}}$ times to trigger the overflow. Consequently, reference counter overflow can be oblivious to security-focused testing techniques such as fuzzing [16, 64]. However, a typical error is a missing counter decrement, which will inadvertently lead
to a memory leak that could be detected.

**Security through better interfaces** It could be argued that many errors are due to poor interfaces that lead to *ad hoc* solutions. Linux reference counters fit this description: they were traditionally implementing using hand-crafted implementations around `atomic_t`. As the experience with `PAX_REFCOUNT` exemplifies (Section 3.2.1), it is not always sufficient to provide better security. Changes must also be intuitive and self-documenting to be acceptable and support future use. However, as `kref` exemplifies (Section 3.1.1), clean and secure interfaces must also accommodate existing needs. The `refcount_t` design takes the best of both: 1) it efficiently prevents overflows, 2) it has a clear use-case without surprising semantics, and 3) it provides a wide API but includes compiler warnings that promote safe use. This assessment is supported by the success of `refcount_t` conversion efforts but also by its incompatibility with bugs. For instance, bugs caused by missing return value checks that lead to reference counter underflows.

**Mixed bounds checking** As discussed in Section 2.3, memory safety is difficult to realize. MPXK takes a mixed approach to sidestep practical limitations, i.e., by using either static pointer-based bounds or dynamically loaded allocation-based bounds. However, allocation-based bounds checking has already been shown problematic [69]. It is likely that such results apply to the mixed approach of MPXK. Nonetheless, trade-offs and similar mixed approaches are often needed for deployable defenses. A useful avenue of research would be to explore how such real-world deployments can be systematically evaluated and compared.

**MPX today** As noted in Section 3.1.2, GCC has dropped MPX support since version 9.1 [67]. Nonetheless, several research projects utilize MPX in various ways but typically only use a subset of MPX instructions. For instance, SGXBounds [100] uses the `bndcl` and `bndcu` instructions to perform bounds checking within Intel SGX enclaves [45]. Due to the restricted address-space of an SGX enclave, SGXBounds can store the bounds within the unused bits of the pointer itself, thus not needing the MPX metadata. In other cases, MPX is used for more coarse-grained enforcement, which allows these approaches to forgo the metadata-use [96, 119, 132, 30].

**Memory safety in restricted environments** MPXK tackles the problem of bounds metadata within the Linux kernel. Some recent projects focus on protecting the kernel (e.g., kCFI [120] and KAISER [74]), but this space is still largely unexplored by the research community. In comparison to userspace, the kernel will introduce new requirements and restrictions for other defenses also. Such restrictions should be researched further. Moreover, other environments will impose different problems. Trusted

---

3lkml.org/lkml/2017/6/27/409
execution environments (TEEs)—which are discussed further in Chapter 4—are a good example; due to their explicit focus on safety-critical programs, their memory safety is of utmost concern. Defenses such as SGXBounds [100] indicate that this space also offers new challenges and opportunities for novel memory safety solutions.
4. Intel SGX side-channels

Intel Software Guard Extensions (SGX) is a hardware feature introduced in the Intel Skylake CPUs [45]. It allows the creation of trusted execution environments (TEEs) called SGX enclaves. An enclave provides three main properties: 1) isolated execution, which prevents other processes from observing its execution state (e.g., CPU registers and enclave memory), 2) sealing, which binds sensitive data to a specific device, and 3) remote attestation, which a remote party can use to verify that specific software is running in an enclave on real SGX hardware. An enclave provides confidentiality and memory integrity within an untrusted environment, e.g., on an untrusted client device or on shared hosting (Figure 4.1).

An SGX enclave is set up by loading it from unprotected memory into enclave memory. The loaded memory, e.g., code and data, is measured to support integrity checks and attestation. To start the enclave, the processor is switched to enclave mode, and the execution transferred to a fixed enclave entry point. An enclave is a started from userspace and has the memory access permissions of the invoking process. It relies on the untrusted OS kernel for scheduling and other system services. However, SGX prevents the kernel and other execution modes from observing an enclave’s execution state. It also ensures that data is encrypted when it leaves the CPU boundary and is written into memory.

Intel states that side-channels are not considered within the threat model of SGX [88]. However, SGX-enabled hosting services are currently offered by several companies (e.g., Microsoft Azure [117], IBM Data Shield [81]). Considering that SGX is used in such environments, side-channels are a concern irrespective of Intel’s envisioned threat model.

In Publication II we investigate a branch-shadowing side-channel on Intel SGX. Like other software side-channels—e.g., cache side-channels—it infers confidential data by observing changes in micro-architectural behavior caused by the processing of that data. Specifically, it exploits the behavior of branching instructions. This makes traditional defensive programming approaches ineffective [4]. Compile-time instrumentation can directly control branch instruction, and therefore, is ideal for addressing
Intel SGX side-channels

Figure 4.1. An Intel SGX enclave is a TEE within an otherwise untrusted system. Remote attestation can be used to establish a trusted communication channel to a remote enclave.

this side-channel (RQ1).

4.1 Background: side-channels and SGX

A side-channel consists of some observable property that depends on a targeted confidential property. For example, the power consumption of a device could be used to leak secret keys from a device [95]. Side-channels can be divided into physical and software side-channels. The distinction can be fuzzy, as software interfaces can give access to physical properties. In this work, we focus solely on software side-channels. They are interesting because they do not require physical access and could be used remotely. An attacker on a virtual hosting platform could use software-only side-channels to attack co-located virtual machines (VMs).

Cache side-channels Cache side-channels are well known. They exploit the CPU’s last-level cache, which is shared between cores [167]. Accessing data in the cache is significantly faster than loading it from memory. This can be exploited to infer what data has been accessed by other processes on the same core. To exploit the cache, first clears it (e.g., by filling it with other data, or flushing it programmatically). If a subsequent data load of the targeted data is fast, can infer that another process has populated the cache by accessing it. Although SGX encrypts memory, the data in the CPU caches remains unencrypted, and so allows cache-based side-channels against SGX [22, 118]. SGX-specific defenses in the literature randomize data [21], reserve a core for the enclave [126], or prevent enclave interruption [143].

Controlled-channel attacks SGX side-channels are not limited to the CPU cache. Because the execution of an SGX enclave is controlled by an unpriv-
Illegal kernel, the kernel itself could be mounting attacks on the enclave. Although the kernel cannot decrypt enclave memory or inspect registers during enclave execution, it 1) knows the code layout because the enclave was loaded from userspace, 2) controls process scheduling and can interrupt the enclave, and 3) controls the memory page tables, including those for enclave memory. Controlled-channel attacks exploit this by selectively marking enclave memory as unavailable in the page table. This causes the enclave to page-fault, which allows $A$ to infer which memory pages are being accessed [166]. Controlled-channel attacks have been demonstrated to leak encryption keys from OpenSSL and Libgcrypt [144]. Later work has shown that page accesses can be observed even without page faults [24], thus circumventing defenses that prevent page faults [143].

**Branch-shadowing side-channel** The branch-shadowing attack exploits the branch prediction unit (BPU) of the CPU [107]. The BPU is used to perform *speculative execution* during *transient execution*. During transient execution, the CPU executes a batch of instructions *out-of-order*. Out-of-order execution allows the CPU to optimize the use of the memory and other resources. However, when transient execution reaches a conditional or indirect branch, the subsequent instruction might not be known. For instance, the target of an indirect function call might be pending a memory load. Speculation is used to predict the outcome of branches so that transient execution can continue. To facilitate this, the BPU keeps a history of prior branch outcomes. The SGX branch shadowing attack exploits the branch target buffer (BTB) that keeps a history of branch targets. Other BPU side-channels target the pattern history table (PHT), which is used to predict whether a (conditional) branch is followed or not [63].

Because behavior changes at run-time, mispredictions will occasionally occur. Mispredicted transient execution must be rolled back. This is not visible on the architectural level; the processor discards the bad state and then computes the correct branch in order to reach the correct architectural state. Results are only committed when they are confirmed (e.g., when all pending memory loads have been completed). A misprediction can be inferred by measuring the execution speed of a branch. Recent Intel CPUs also provides the last branch record (LBR) performance counter that can be enabled to log mispredictions [83].

Because branch behavior is often data-dependent, mispredictions can be used as a side-channel. BPU side-channels can, for instance, leak RSA-keys [4] and break ASLR [62]. BPU internals and functionality are proprietary. Nonetheless, experimental results have shown that BTB is indexed based on specific bits from the branch instruction’s memory address. $A$ can thus create a *shadow branch* that shares the BPU history $H$ with another targeted instruction (Figure 4.2). Because history is shared between different processes on the same core, the shadow branch can be
Intel SGX side-channels

Figure 4.2. can infer enclave branching behavior dependent on confidential data by observing how it affects the BPU history. The branch history \( H \) is first primed with a known state ( ), then the attacked enclave is allowed to execute ( ). The enclave execution will, depending on the confidential data, change the branch history \( H \). will then interrupt the enclave before ( ), enable LBR, and re-execute the shadow branch to see if \( H \) was affected by the enclave ( ).

in another process. This also applies to branches within SGX and can be used as a side-channel to monitor enclave execution [107].

The branch-shadowing attack To mount an attack, first analyzes the enclave code and identifies a branch instruction to target. Although the enclave is in encrypted memory, can observe the loading process and thus knows the memory layout of the code sections. then constructs a shadow branch that uses the same BPU history \( H \) as the target branch (Figure 4.2). Before launching or resuming the enclave, primes \( H \) to a known state by repeatedly executing the shadow branch. then allows the enclave to briefly execute before interrupting it again. After enabling the LBR performance counter, executes the shadow branch again. If a misprediction is reported in the LBR, can infer that \( H \) was changed by enclave execution. The execution flow leaks information on processed data and has been shown sufficient to leak RSA-keys from an SGX enclave [107].

Initial branch-shadowing defenses Zigzagger [107] is an initial defense against branch-shadowing and depends on ’s inability to perform fine-grained interruptions of an enclave. Conditional branches are first converted to indirect branches. The indirect branch targets are setup using conditional moves (\( \text{cmov} \)). The execution of indirect and unconditional branches can still be inferred via shadow branches. To prevent this Zigzagger executes all related branch instructions before reaching the intended target. The security of this scheme requires that ’s inability to perform fine-grained interruptions to distinguish the meaningful branches from the decoys. Unfortunately, later work shows that enclave execution can be controlled at single-instruction granularity [157]. This allows to observe each branch separately, thus breaking Zigzagger.
4.2 Results: preventing SGX branch shadowing

Fine-grained branch-shadowing can break randomization, such as ASLR [62], including SGX-specific randomization techniques [138, 157]. Defenses that target timing side-channels are similarly ineffective against branch shadowing. Zigzagger is promising but relies on a weak adversary model [157, 107]. In Publication II we propose a novel approach that thwarts a strong adversary mounting a branch-shadowing attack. Our approach relies on compile-time instrumentation to randomize program control flow. To minimize increased attack surface and allow attestation, our approach only randomizes small sections of the code.

We first re-use the idea of implementing branches using conditional moves. All branches are then routed via a randomized trampoline, which is a small piece of code that immediately branches (or jumps) to another memory address. Branches into the trampoline are always followed, and thus reveal no information to A. Because the trampoline locations are randomized, A is forced to guess the location of the trampoline to shadow. Furthermore, because the BPU history is of limited size, A only has a limited number of guesses. With enough entropy, A cannot reliably shadow the correct address before it is overwritten, thus preventing the attack.

4.3 Discussion: side-channel challenges

The branch-shadowing attack relies on obscure—and proprietary—runtime behavior of the CPU. It cannot be prevented by common side-channel resistant programming practices. For instance, balancing the cycle counts of different branches is ineffective because the attack relies on the branch instruction itself. Branch-instruction placement also depends on compiler optimizations, such as inlining and loop unrolling. This alone suggests that branch shadowing is best tackled by the compiler.

However, side-channels cannot be viewed in isolation. Generating a program with only mov instructions would hide all branches, but also increase memory use and code size, thereby making cache side-channels easier to execute [57]. This also applies to our work, which only affects the branch-shadowing attack, and must be combined with other defenses to provide full protection. Moreover, A is not limited to using one attack. For instance, by monitoring memory accesses, A could limit the entropy afforded by randomization techniques, including our defense. Any side-channel defense should be compatible and integrated with complementary defenses, and avoid introducing new vulnerabilities. The complexity of such interactions suggests that a systematic approach is needed.

Spectre attacks The stage is drastically changed with transient execution attacks [27], e.g., Meltdown [110] and Spectre [94]. Transient execution
attacks have, unsurprisingly, also been demonstrated on Intel SGX [98, 32, 156]. Spectre, in particular, has a marked resemblance to branch-shadowing as it also exploits the BPU. In contrast to side-channel attacks that monitor changes caused by the victim process, Spectre attacks manipulate the micro-architectural state to affect the transient execution of the victim process. To perform a Spectre attack, $A$ would first train the BPU to mispredict transient execution so that it accesses some confidential code. Due to the misprediction, the transient state is eventually rolled back. To leak the data, $A$ uses a covert channel to transmit the data out of the transient state. The CPU-caches are affected by transient execution even when the execution is rolled back later. This can be exploited by ensuring that the transient execution, before rollback, performs some data access that depends on the confidential data. Finally, $A$ just needs to probe the associated cache to retrieve the confidential data [167].

The Spectre attacks reverse the role of micro-architectural side effects. Traditional side-channel defenses hide observable differences in the micro-architectural state after confidential data is processed. But Spectre attacks focus on manipulating the prior state in order to cause the transient access of confidential data. This requires different defense strategies. Our branch-shadowing defense prevents a BPU side-channel but not Spectre. In fact, the added static trampoline entry-points provide more branches to manipulate in a Spectre attack. But Spectre defenses are similarly ineffective against branch-shadowing. For instance, based on our evaluation in Publication II, speculation fences [84] that prevent speculation beyond a specific point, do not prevent branch shadowing.

**Future outlook** Because side-channels (and transient execution attacks) are nuanced, depend on obscure micro-architectural behavior, and evolve quickly, they require systematic and automatic defenses. Side-channels often rely on hardware features, not bugs. These features are often necessary for performance; e.g., caching and speculation substantially improve execution speed. In the case of SGX, one could envision hardware solutions, such as conditionally-updated or separate micro-architectural states [65, 91]. But in the short-term, hardware fixes are unavailable. Intel has introduced microcode updates that mitigate Spectre attacks on SGX [84]. But these are not effective against all attacks. For instance, based on our evaluation presented in Publication II, the indirect branch restricted speculation (IBRS) feature does not prevent SGX branch shadowing. General hardware fixes are even more unlikely to emerge due to performance constraints. For instance, cache side-channels have been known for decades and are an accepted cost of performance. As efficient protection seem unlikely to appear, this suggests that transient execution attacks are here to stay.

Software-based solutions are needed, with or without upcoming hardware assistance. Meanwhile, solutions must allow conditional protection to limit performance impact. To manage complexity, side-channel defenses
cannot be built in isolation. Interactions among different solutions must be recognized and accounted for. In practice, this requires systematic and automated instrumentation support. Although binary-only instrumentation might be possible, compile-time instrumentation is more suited to modify code structure, e.g., by modifying the CFG. In conclusion, we need compiler support for 1) conditional protections that only protect specific data, 2) programmer annotations that mark sensitive data, and 3) hardware-specific defenses that minimize performance overheads.
Full memory safety solutions so far have proven inefficient or been built around custom hardware. Successful and widely-deployed defenses, in contrast, have a more narrow scope. They also typically target different stages of an attack: for instance, stack canaries [47] do not prevent memory errors, but instead, allow the detection of stack corruption before a corrupted return address is used. Other defenses prevent memory errors directly. Stateless CFI solutions prevent the injection of executable memory [116, 130]. Stateless CFI solutions verify function call targets without addressing underlying memory errors [2]. None of these approaches are perfect. But they are efficient and significantly decrease the attack surface.

In this spirit, pointer integrity only aims to prevent the corruption of pointers [101]. It does not provide full memory safety; it only guarantees that pointers cannot be corrupted. Nevertheless, this is powerful because pointer corruption can be used to 1) create an arbitrary-write primitive [168], 2) redirect program control flow [141], and 3) mount non-control data attacks [33]. Pointer integrity can thus prevent a large class of attacks. Even when an attacker $A$ has an arbitrary-write primitive—e.g., using an unbounded indexing error—powerful attacks such as ROP [141] and DOP [79] are prevented if the integrity of pointers is guaranteed.

In Publications III and V, we explore how the recent ARMv8.3-A Pointer Authentication (PA) extension can be used to achieve pointer integrity. We investigate weaknesses in PA-based defenses and show how to mitigate them. In Publication IV, we demonstrate the general nature of PA by using it to harden stack canaries.

### 5.1 Background: ARMv8.3-A pointer authentication

The PA extension was introduced in the ARMv8.3-A architecture released in 2017 [133, 8]. PA provides hardware support for “signing” and verifying pointers with an embedded MAC, called the pointer authentication code (PAC). Because the PAC is generated by hardware, PA has high perfor-
Figure 5.1. PA avoids both metadata and changing pointer size by embedding the PAC in the sign-extension bits of a pointer.

mance. Evaluation of a candidate algorithm, QARMA, indicates that a PAC can be calculated in only four cycles on a 1.2GHz core [13].¹ This makes PA feasible for run-time protection of production software. Because the A-class architecture targets full-fledged operating systems and is used by most contemporary mobile phones, PA is expected to be widely deployed.

PAC construction The PAC is calculated using a tweakable MAC algorithm. It is based on a 128-bit hardware-protected key, an instrumentation-dependent 64-bit modifier as the tweak, and the virtual address (VA) of a pointer. PACs are created and verified with explicit instrumentation using the new pac and aut instructions. The pac / aut instructions have several variants, each of which uses one of five available keys. Two keys protect code pointers, two data pointers, and one is for generic use. When a pointer is signed, the resulting PAC is stored in the sign-extension bits of the 64-bit pointer (Figure 5.1). Its size is between 3 and 31 bits, depending on how many bits are reserved for the VA and whether 8-bits are reserved for memory tagging [8]. On default AArch64 Linux configurations, the PAC is 16 bits [133]. PA also provides the pacga instruction that uses the generic key to calculate a 32-bit PACs over an arbitrary 64-bit value and the given 64-bit modifier.

Current PA support The Linux kernel supports—since v5.0—userspace PA by initializing PA keys at process exec [109]. PA-based return-address protection is supported by both the GCC [70] and Clang [35] compilers.² On ARMv8 architectures, the function return address is stored in a specific register, LR, at function entry. Non-leaf functions must then store it on the stack to allow nested function calls that overwrite LR. This leaves the return address vulnerable to corruption and allows ROP attacks on ARM [97]. The returned address is signed before it is stored on the stack and again verified before function return.

The PA modifier As Linux userspace PA keys are process-specific, they are constant throughout the process lifetime. This binds the signed pointers to a specific process but allows pointers within a process to be swapped. We call this attack a reuse attack, as it reuses previously signed pointers in a

¹ARM does not mandate the use of QARMA, but mentions it as an alternative [8].
²On AArch64 targets with PA, return-address protection can be enabled with the -msign-return-address flag, or the -mbranch-protection flag that includes other defenses.
different context. But the PAC value also depends on the instrumentation-controlled modifier. The modifier value can be used to constrain reuse attacks by binding a pointer to a specific context or property. The return-address protection of GCC / Clang, for instance, uses the stack pointer (SP) value as the modifier. The SP is convenient as its value changes during program execution but is guaranteed to be the same at function entry and exit. This narrows the scope of reuse attacks without the need to explicitly keep track of the modifier value.

**PA error detection** To verify a signed pointer, the PAC is re-calculated. If the resulting PAC matches the PAC embedded in the pointer, verification succeeds. On success, the PAC is stripped and replaced with the sign-extension bits. On failure, the PAC bits are first replaced with the sign-extension bits, after which the pointer is invalidated by flipping a specific high-order bit. Failure does not cause an immediate fault. However, when the pointed-to address is translated by the memory management unit (MMU)—for instance, during instruction fetch on return—the MMU detects the invalid bit and issues an address translation fault.

### 5.2 Results: pointer authentication and stack safety

In this dissertation, I explore PA through three publications: 1) in Publication III, we show how to mitigate PA reuse attacks, and more broadly, how to realize run-time type checking with PA; 2) in Publication IV, we demonstrate the versatility of PA by using it to harden stack canaries [47]; and 3) in Publication V, we show that PA can provide precise return-address protection similar to hardware-based shadow stacks [82].

**Type-checking with PA** The modifier used for PACs should ideally be unique to a specific pointer value; this would prevent any reuse attacks. But the modifier must also be available both when the pointer is signed and when it is verified. A randomly assigned modifier—or a nonce—would require the modifier to be securely tracked or otherwise associated with the correct pointer. If such a scheme were possible, it could as well secure the pointer itself without the need for PA. In Publication III, we propose to bind the modifier to the pointer type. This affords two powerful properties: 1) it prevents the injection of arbitrary pointers by using PA; and 2) it enforces run-time type checking, even in the presence of reuse attacks.

Our prototype implementation, PARTS (Publication III), is implemented on LLVM 8.0. It supports run-time type-checking for both code and data pointers. It also features an improved return-address protection scheme that binds each return address not only to the SP value but also to a function-specific identifier. This drastically decreases the scope of reuse attacks but does not completely prevent them. Our benchmarks show that
PARTS incurs a very low overhead (< 0.5%) when used to protect code pointers and return addresses. This prevents ROP and other control-flow attacks with a minimal performance overhead.

**Generating stack canaries with PA** Stack canaries (Section 2.2.1) are a low-cost defense. They continue to be widely supported by compilers since their conception in 1998 [47]. But stack canaries suffer from known weaknesses (e.g., using a program-wide reference value). Proposed hardening schemes (e.g., DynaGuard [131], DCR [76], and polymorphic canaries [160]) have not been enabled by mainstream compilers, presumably because the performance cost is too high to be justifiable. PA return-address protection already functions as a stack canary. In Publication IV we build upon this realization and design a more secure stack canary scheme that also incurs negligible performance overhead.

Our design uses PA to generate canaries without depending on in-memory reference values. When needed, we also generate multiple canaries to detect precise overflows that could avoid detection when only one canary per stack frame is employed. Each added canary is constructed as a signed pointer to the previous one; this allows easy verification of the canaries and ensures that each canary within a stack frame is unique. All canaries are generated using a function-specific modifier value; this binds the canaries to their respective functions. Evaluation of our prototype, PCan, shows that these additions incur only negligible overhead, despite the substantial gain in security compared to traditional canaries.

**PA-based return-address protection** An ideal PA-modifier scheme is frustratingly elusive. Nonetheless, in Publication V, we show that such a scheme is possible for return-address protection. Our design completely eliminates reuse attacks and limits possible attacks to guessing. The intuition behind our solution is that a chained-MAC of return addresses uniquely identifies a call flow. With PA, we can efficiently generate the chain and then use it to verify return addresses (Figure 5.2). Only the final value is needed to verify the whole chain of return addresses. Because each function only adds and removes the topmost value, no extra PAC calculations are needed compared to prior PA return-address protections.

We further refine the design by using signed return addresses, \( \text{arest}_i \),
ARM Pointer authentication

defined as:

\[
\text{signedReturnAddress}_i \gets \text{sign(\text{returnAddress}_i, \text{signedReturnAddress}_{i-1})}
\]

The previous signed return address, \(\text{signedReturnAddress}_{i-1}\) is stored on the stack. The current \(\text{signedReturnAddress}\) is always kept in a register. To support this, the calling convention is changed such that the \(\text{signedReturnAddress}\) is passed to the callee. Although this technically breaks the application binary interface (ABI), our prototype implementation uses a callee-saved register to pass \(\text{signedReturnAddress}\). This allows functional compatibility with uninstrumented code, but still weakens the security guarantees if uninstrumented code is called.

Because the intermediate \(\text{signedReturnAddress}\) values are exposed in memory, \(\ensuremath{\mathcal{A}}\) could attempt to find collisions. To prevent this, we additionally use PA to create a masking value that hides the PAC before it is written into memory. This prevents \(\ensuremath{\mathcal{A}}\) from recognizing collisions. The only remaining attack is guessing, which has a success probability dependent only on the PAC size. Evaluation of our prototype implementation, PACStack, indicates overheads of less than 1\%, or less than 0.5\% without masking.

5.3 Discussion: beyond pointer authentication

Our work on PA shows that it can be used to implement novel and powerful security schemes. Both PARTS and PACStack (Publications III and V) protect pointers. By providing the \texttt{pacga} instruction ARM, invites uses that go beyond pointers.\(^3\) PCan starts exploring this space (Publication IV), but other directions are likely possible. A construction similar to PACStack could, for instance, be used to protect data structures other than the program stack.

Randomly assigned PA modifiers  PA security depends on two things: the hardware-protected keys, and the instrumentation-chosen modifier. Current approaches instantiate the keys at process startup. This prevents the injection of arbitrary pointers and effectivelly binds the signed pointer to the process context. Further narrowing of context is left to the modifier. An ideal solution would be akin to nonces. But nonces need to be tracked. This suggests that practical solutions are limited to static modifiers or naturally tracked values such as the SP value.

Static modifiers  Statically assigned modifiers that are unique to a particular pointer value would be ideal. If a pointer has multiple values during execution, this is not possible. Even when the pointer, at run-time, will have only one value, this might not be detectable by static analysis. Static

\(^3\)This is also explicitly mentioned in the ARM documentation [8].
read-only pointers are a notable exception as they could use a pointer’s storage address as its modifier. Because no other pointers are written to the same address, the reuse attack is eliminated. For instance, C++ virtual tables—which contain pointers to the virtual functions of a class—are static and could be protected using this method. But in most cases, this approach has limitations familiar from stateless CFI.

**PA as a CFI mechanism** As discussed in Section 2.2.2, CFI enforces that a specific call site always targets a function belonging to the correct equivalence class (EC) [2, 3]. Because the EC is typically more inclusive than necessary, such policies can be circumvented [72, 29]. PA modifiers are different; they are not only used at call or dereference to verify, but they are also used to sign the pointer. Any ECs with intersecting pointers must, therefore, be merged. This suggests that PA modifiers based on static analysis are ineffective in preventing the exploitation of reuse attacks. Note that, in contrast to stateless CFI, PA always prevents injection of arbitrary pointers; this consideration only applies to reuse attacks.

**Reuse attack prevalence** Our work considers reuse attacks on PA. But do reuse conditions—for instance, when using the SP as a modifier—occur in real programs? Based on our preliminary evaluation, the answer is yes. This is, perhaps, somewhat counter-intuitive. But because stack-frames are aligned, different stack-frames often have the same size despite containing different variables.

A natural follow-up question is whether reuse attacks are exploitable in practice. More research is needed, but the answer is likely yes. In the context of CFI, prior research [137, 51, 72, 29] suggests that reuse attacks can be exploited. Evaluating CFI effectiveness is notoriously hard [165]. Nonetheless, metrics based on the EC count [26] could be applicable to PA. But CFI metrics do not model the need to generate reusable pointers. To accurately model reuse attacks, an analysis would need to consider the possible call flows that generate reusable pointers. A full analysis is likely not feasible, and so new heuristics are needed.

**Upcoming hardware-security primitives** PA is not the only ARM security extension being rolled out: ARMv8.5 is adding the new Memory Tagging Extension (MTE) and BTI extensions. MTE is a memory tagging scheme, which allows userspace programs to tag pointers and memory regions. The hardware then ensures that tagged memory is accessed only with correctly-tagged pointers. BTI enforces that indirect branch instructions, e.g., indirect function calls, always have a valid target, e.g., a function entry point. Coupled with PA, these constitute a varied and powerful set of primitives that will likely allow novel defenses. Research is needed to understand the interactions between different security mechanisms better.

RISC-V has already proven itself to be fertile ground for exploring new...
ARM Pointer authentication

hardware features, for instance, run-time scope enforcement [123], control-flow attestation [53], data-flow isolation [146], and tagged memory [148]. It could also be used to combine different extensions and explore new emergent properties.

A related question is whether similar policies are possible using different hardware extensions. For instance, can PA-based policies be implemented using Intel hardware? Prior research has demonstrated that policies similar to PARTS (Publication III) can be implemented using hardware-accelerated cryptographic primitives on Intel CPUs [113]. But this approach has significantly higher performance overhead. PACStack (Publication V) and hardware-based shadow stacks [25] are other examples, where both mechanisms achieve very similar performance and security properties. These examples are likely not isolated.
6. Discussion and Conclusion

6.1 Preventing memory errors in unsafe languages

One of my objectives in this dissertation has been to use hardware-assisted instrumentation to prevent memory errors. Prior work has approached this challenge from several directions in the literature (Section 2.3). One successful approach is the use of run-time checks, typically added at compile-time. Currently deployed approaches are mostly software-based. For example, the stateless CFI supported by Clang [37]. Unfortunately, many defenses in the literature are impractical to deploy, either due to performance considerations or compatibility issues. A recent survey of CFI research found serious compatibility problems—for instance, with exceptions and dynamic linking—in most designs [165]. Nonetheless, there are several fruitful research directions, including fuzzing, static analysis, and hardware-assisted memory safety.

**Fuzzing** An alternative to run-time checks is to focus on testing. Fuzzing is a technique for automated testing which has gained considerable traction [147]. It creates test cases either randomly [16, 64] or uses techniques such as symbolic execution to improve test-generation efficiency and get code coverage feedback [71]. However, errors must be both triggered and detected. As seen with reference counters (Chapter 3), some error types can be inherently hard to trigger using fuzzing techniques. Even when triggered, a memory error might not be detected unless it causes a crash.

To improve the detection rate, fuzzed programs can be instrumented with additional run-time memory safety checks. Because fuzzing is performed during testing, it does not affect the performance of deployed programs. This permits the use of tools such as AddressSanitizer during fuzzing [11]. Nevertheless, resources spent on fuzzing are not free, and so the performance of fuzzed binaries affects cost of development.
**Static analysis**  Static analysis can be used to detect errors during development. Unfortunately, many types of static analysis are NP-complete problems [104]. In particular, memory safety analyses have been shown to be undecidable [136, 169]. Therefore, analyses that detect memory errors are often either probabilistic [17] or focus on specific error types [66, 58]. For instance, the Clang compiler offers multiple static analysis tools that can detect different errors [36]. Many memory safety defenses rely on static analysis for instrumentation. To guarantee run-time functionality, such defenses must be conservative when analysis results are incomplete. Consequently, security policies based on static analysis are often needlessly permissive. For instance, see Section 2.2.2 on stateless CFI.

Run-time security mechanisms could be incorporated into analyses to improve their accuracy and efficiency. A security mechanism such as PARTS (Section 5.2, Publication III) restricts the possible run-time values of a pointer. By being made aware of added security properties, static analyses could be made both faster and more precise. Allocation-based spatial memory safety is another example. Although the allocation bounds cannot prevent all attacks [112], they isolate different memory regions, and so allow local reasoning of memory accesses. The opposite approach is already true: deployed defenses—for instance, stack canaries in Clang [35]—often minimize performance overhead by using static analysis to omit unnecessary instrumentation. Exploring tighter integration of security policies or mechanisms into static analyses will likely be worthwhile.

**Holistic hardware-assisted security**  The recent influx of new memory-safety features in off-the-shelf hardware is a central topic of this dissertation (Publications I, II–V). My work has focused primarily on the run-time security aspects, which undoubtedly are essential. However, run-time security, fuzzing, and static analysis are not mutually exclusive. Future research should focus on their complementary aspects and exploring closer integration. Fuzzing, for example, could directly benefit from hardware-assisted memory safety. Because it is a time-intensive activity, performance gains afforded by hardware-assistance directly transfer to fuzzing. As discussed above, static analyses would also benefit from closer integration with hardware-assisted memory safety guarantees. By following such research directions, hardware-assisted memory safety can improve the whole software life cycle. This would increase the value of such features, and consequently, increase the incentive to introduce new security features in upcoming hardware.

### 6.2 Memory safe languages

When discussing memory safety in C / C++, one cannot avoid the implications of memory-safe languages. Of these, the Rust [114] programming
Discussion and Conclusion

language is perhaps the most notable. As an example, I will discuss Rust, but the examination also applies to other memory-safe languages. The devil's advocate would argue that it would be better if developers switch to using Rust rather than trying to protect C / C++. However, as discussed in Section 1.1, C / C++ is not going anywhere anytime soon. Perhaps it is more interesting to ponder what hardware-assisted memory safety research and Rust can offer each other. The run-time security of Rust depends on two things: 1) extensive static analysis that detects and prevents memory errors at compile-time, and 2) run-time checks, for instance, to validate array bounds. Consequently, a program comprised of safe Rust code offers strong security properties.

Mixed code But Rust also supports unsafe code, that is, code that allows unsafe operations (e.g., raw pointers). This allows efficient implementation of low-level code, such as device drivers. But unsafe code breaks the security guarantees. A Rust library that uses unsafe behavior could compromise the security of safe code sections. In some cases, such libraries can be proven not to affect the safety of safe code sections [90]. But in the general case, this might be impossible. Moreover, the Rust foreign function interface can also be used to interact with C / C++ libraries [93].

To maintain the security of safe Rust code, any unsafe Rust and C / C++ libraries could be compartmentalized. Process isolation can be used to isolate C / C++ libraries, but this incurs a high overhead [103]. Meanwhile, hardware-assisted security could be used to: 1) guarantee the run-time safety of some unsafe Rust operations, 2) mitigate the impact of unsafe Rust and C / C++, or 3) provide performant in-process isolation of C / C++. Efficient in-process isolation can be achieved using capability machines such as CHERI [154]. Moreover, recent works have also demonstrated that this can be achieved using commodity hardware, such as Intel MPX [96] or PKU [155]. It is likely that similar properties can be realized with security-primitives in ARM architectures.

Compile-time performance As Rust aficionados will attest to, Rust can achieve run-time performance on par with C / C++. Improved run-time performance comes at the cost of degraded compile-time performance; Rust compilation times can be orders of magnitude longer than similar C / C++ code. Compile-time overheads are a result of comprehensive static analysis. As mentioned above, analyses could benefit from modeling new security properties afforded by hardware-assistance (Section 6.1). It might also be possible to omit compile-time verification of security invariants that can be guaranteed by the hardware. Overall, Rust and C / C++ will likely continue to exist in a shared ecosystem. Consequently, research in this area is not a zero-sum game between safe and unsafe languages.
6.3 Understanding memory errors and exploitability

Memory errors are a fuzzy problem. Various definitions abound (Section 2.3). Theoretical definitions of memory safety exist, but can be too expensive, too restrictive, or impossible to apply in the general case. For example, large-scale programs might be too complex for complete analysis or have unavoidable dependencies to code that cannot be proven safe. Consequently, such definitions are not practical beyond relatively small security-critical software (e.g., cryptographic libraries). Approaches that permit partitioned [159] or conditional analyses [41, 15] could bridge this gap.

Compartmentalization Memory safety is often an either-or proposition. This limits the usability of memory safety analyses in environments that must use unsafe code. One solution is to compartmentalize such code and libraries into their own security domain. Efficient solutions to achieve this have been demonstrated using, for instance, CHERI [154], ARM memory domains [170], Intel MPX [96], and PKU [155].

Weird machines and exploitability As memory errors often cannot be completely avoided or proven absent, it behooves to ask when a memory error is exploitable. But analyzing the exploitability of memory errors is a difficult task [59]; the evaluation of CFI is a prime example of this difficulty (Section 2.2.2). The concept of weird machines has been proposed to facilitate such analysis. Intuitively, a weird machine is the finite state machine (FSM) that emerges when a program reaches an unintended state [23, 158], for example, as a result of a memory error. In this model, attacks like ROP [141] execute on a weird machine. The machine is defined as the intended FSM extended with the states and state transitions induced by ROP gadgets. Indeed, research suggests that weird machines could—not only have predicted ROP and DOP [79] attacks—but also made CFI weaknesses [137, 51, 72, 29] immediately apparent [59].

Analyzing unsafe code In this dissertation, I only scratch the surface of exploitability and static analysis of memory errors. Nonetheless, the presented publications raise questions of exploitability. For instance: 1) how does pointer- and allocation-based memory bounds checking differ in terms of security (Publication I), 2) how different PA-policies affect the exploitability of reuse attacks (Publication III), and 3) how can we design meaningful side-channel defenses within a rapidly evolving threat landscape (Publication II)? My results suggest that more research is needed. In particular, models that capture the interactions between unsafe and safe code would allow practical analysis of codebases that cannot fully adhere to strict standards or guarantee the memory safety of all components.
6.4 Conclusion

In this dissertation, I have demonstrated how to use security mechanisms readily available in off-the-shelf hardware to realize strong and performant memory safety defenses. My work on mitigating the branch-shadowing attacks on Intel SGX (Publication II) and the Spectre attacks (that surfaced at the same time) are a prime example of the complexity involved in mitigating run-time attacks. Such complexity is not limited to mystifying side-channels; memory errors, and in particular their exploitation, are notoriously hard to evaluate. However, hardware-assisted security features can provide guarantees beyond what is feasible in software alone. The benefits of hardware-assisted memory safety are not restricted to run-time. Other techniques such as memory-safe languages, fuzzing, and static analysis can benefit from advances in hardware-assisted memory safety. My work approaches hardware-assisted memory safety by demonstrating how to use Intel MPX (Publication I) and ARMv8.3-A PA (Publications III–V). These are not prototype-hardware built to accommodate research; they are deployed in widely-available commodity hardware. This allows greater opportunities for my work to move beyond the sphere of academic research to protect real systems in use today.


Bibliography


[109] Linux_5.0 — Linux Kernel Newbies. 2019. URL: https://kernelnewbies.org/Linux_5.0 (visited on 11/14/2019) (cit. on p. 52).


Bibliography


Bibliography


Errata

Publication III

Pointer Authentication instruction compatibilities incorrectly listed in Appendix C. Correct table is in Publication V.