Preface

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

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.

I Elena Reshetova, Hans Liljestrand, Andrew Paverd, N. Asokan. Towards Linux Kernel Memory Safety. *Software: Practice and Experience*, December 2018.

II Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution, SysTEX '18*, Toronto, ON, Canada, pages 22–47, October 2018.

III Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea, Jan-Erik Ekberg, N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, pages 177–195, August 2019.

IV Hans Liljestrand, Zaheer Gauhar, Thomas Nuyman, Jan-Erik Ekberg, N. Asokan. Protecting the stack with PACed canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution, SysTEX '19*, Huntsville, ON, Canada, 6 pages, October 2019.

V Hans Liljestrand, Thomas Nyman, Lachlan Gunn, Jan-Erik Ekberg, N. Asokan. PACStack: an Authenticated Call Stack. *Submitted*, 20 pages, August 2019.

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 Authentication 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.

VI Mika Juuti, Christian Vaas, Ivo Sluganovic, Hans Liljestrand, N. Asokan, Ivan Martinovic. STASH: Securing transparent authentication schemes using prover-side proximity verification. *Proceedings of the IEEE International Conference on Sensing, Communication and Networking*, 2017.

VII Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, N. Asokan. Late Breaking Results: Authenticated Call Stack. *Proceedings of the 56th Annual Design Automation Conference*, 2019.

VIII Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Danfeng (Daphne) Yao, Trent Jaeger, N. Asokan. Exploitation techniques and defenses for data-oriented attacks. *Proceedings of the IEEE Cybersecurity Development*, 2019.

List of Figures

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

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 `W⊕X` 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 powerful 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.

Table 1.1. Publications and corresponding research questions.

Publication		**RQ1**	**RQ2**	**RQ3**
Publication I	(Chapter 3)	✓	✓	✓
Publication II	(Chapter 4)	✓	✗	✗
Publication III	(Chapter 5)	✓	✓	✗
Publication IV	(Chapter 5)	✓	✓	✗
Publication V	(Chapter 5)	✓	✓	✗

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 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.

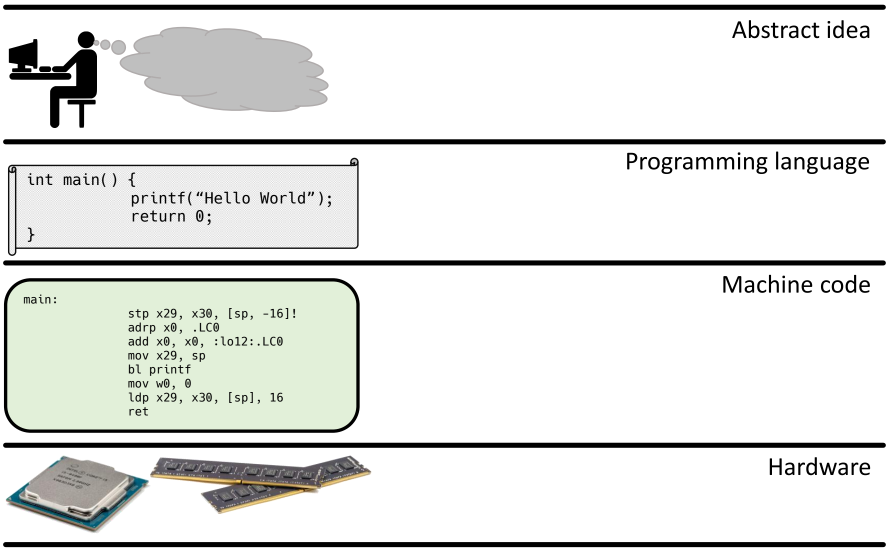
2. Background

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.[1] 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.



```
int main() {
            printf(“Hello World”);
            return 0;
}
```

```
main:
            stp x29, x30, [sp, -16]!
            adrp x0, .LC0
            add x0, x0, :lo12:.LC0
            mov x29, sp
            bl printf
            mov w0, 0
            ldp x29, x30, [sp], 16
            ret
```

Figure 2.1. A computer program can be viewed from three different layers: the source programming language, the machine code that is given to the CPU, and the hardware that executes the computations described by the machine code.

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 languages 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

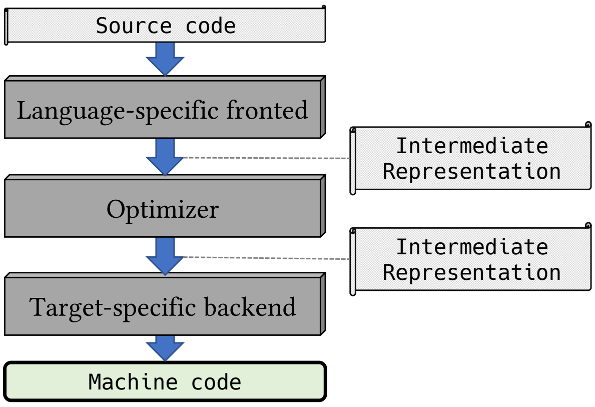

Source code
Language-specific fronted
Intermediate Representation
Optimizer
Intermediate Representation
Target-specific backend
Machine code

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 come 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 $\mathscr{A}$ 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, $\mathscr{A}$ 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 instruction 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. $\mathscr{A}$ can thus not only inject the machine code but also corrupt the return address such that it points to the injected code.

$\mathtt{W} \oplus \mathtt{X}$ policies $\mathtt{W} \oplus \mathtt{X}$ 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 $\mathscr{A}$ from executing the injected code. Today, $\mathtt{W} \oplus \mathtt{X}$ 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, $\mathscr{A}$ 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 $\mathscr{A}$ to effectively call arbitrary functions. Moreover, because the stack grows towards lower addresses, $\mathscr{A}$ can corrupt multiple stack-frames. $\mathscr{A}$ 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.

$\mathscr{A}$ 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, $\mathscr{A}$ 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. $\mathscr{A}$ 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. $\mathscr{A}$ 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, $\mathscr{A}$ 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 hardening 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. $\mathscr{A}$ 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, $\mathscr{A}$ 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

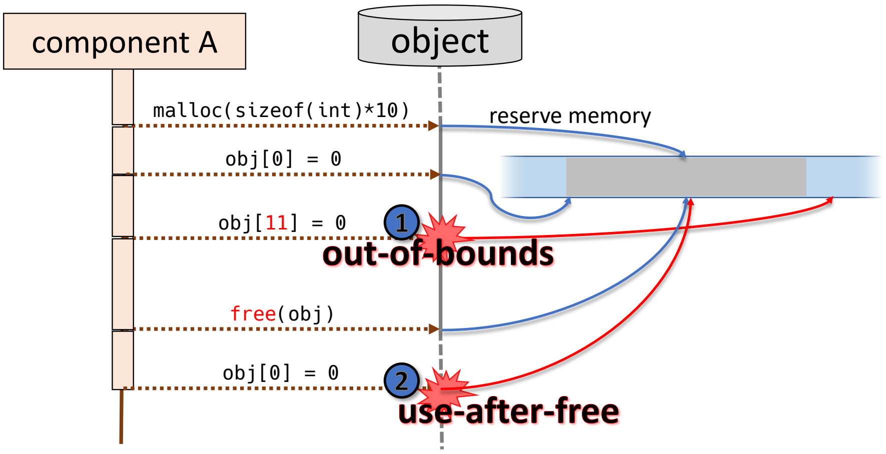


Figure 2.3. Memory errors can be either temporal (e.g., an out-of-bounds write ❶) or spatial (e.g., use-after-free error ❷).

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.

2.3.2 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].

2.3.3 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 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 $\mathscr{A}$ 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, $\mathscr{A}$ typically attacks a process from within it. $\mathscr{A}$ 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, $\mathscr{A}$ 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 $\mathscr{A}$ 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 `INT_MAX` will be set to `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 error could allow the counter to be incremented without allocating resources. This could allow $\mathscr{A}$ 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 `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 `kref` type, which is designed for reference counting and implemented with `atomic_t` [99, 115]. Unfortunately, `kref` is sparsely used, leading to a multitude of hand-crafted reference counter implementations based on `atomic_t`.

Reference counters in the Linux kernel often serve multiple purposes. For instance, the `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.

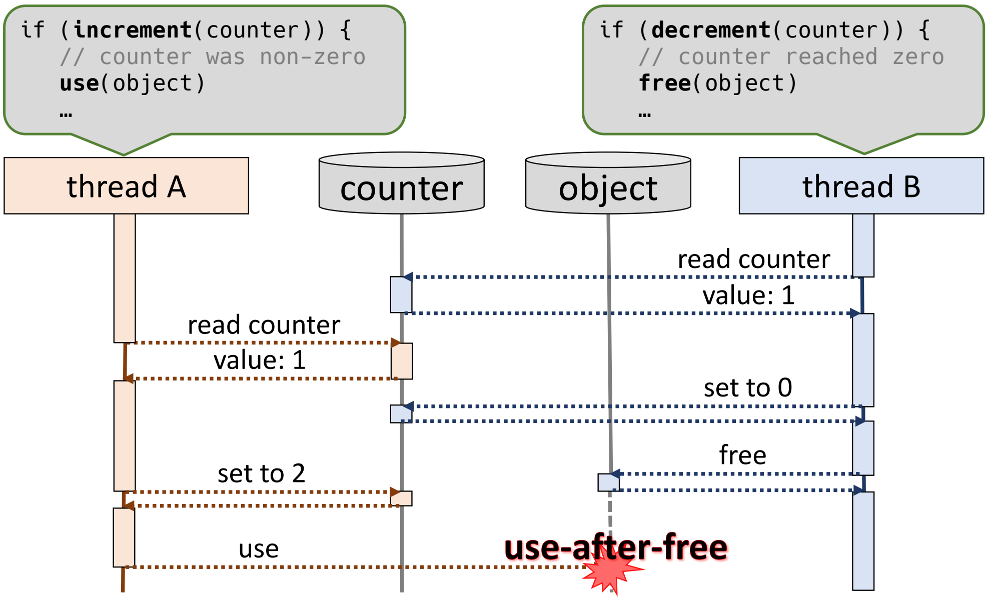


Figure 3.1. Non-atomic reference counters could result in an object being freed by a thread **A** while concurrently being acquired by another thread **B**.

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.

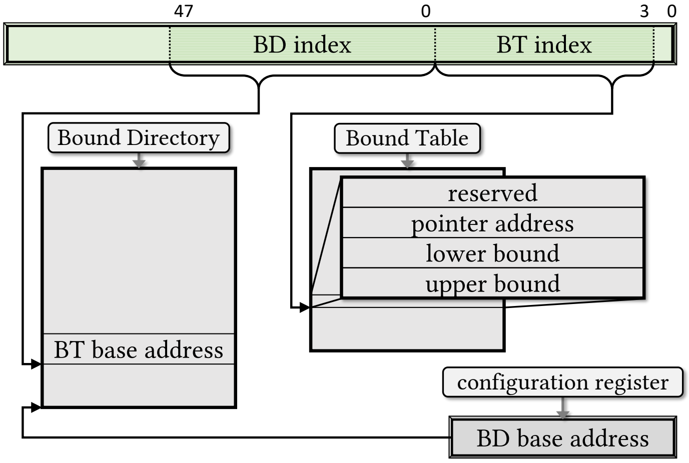


Figure 3.2. MPX bounds metadata uses a two-level mapping based on a pointer's address.

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` / `bndldx`. 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.[3]

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

[3] lkml.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

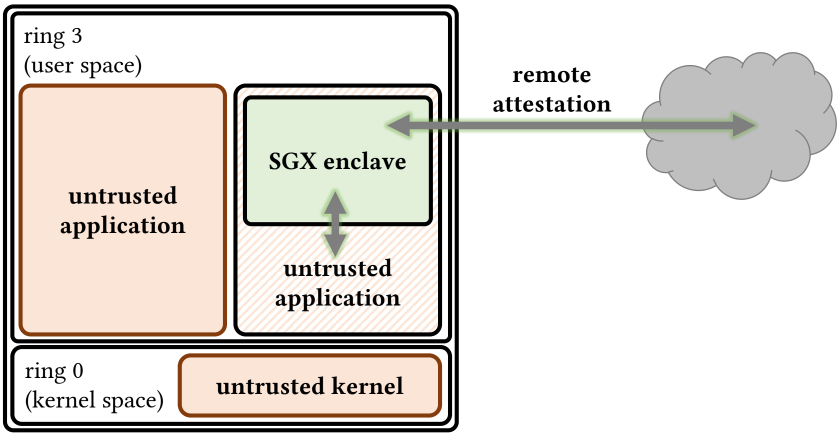


Figure 4.1. An Intel SGX enclave is a TEE within an otherwise untrusted system. Remote attestation can be used to establish a trusted communication channel to a remote enclave.

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 $\mathscr{A}$ 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, $\mathscr{A}$ 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, $\mathscr{A}$ 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 unprivileged 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 $\mathscr{A}$ 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. $\mathscr{A}$ 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

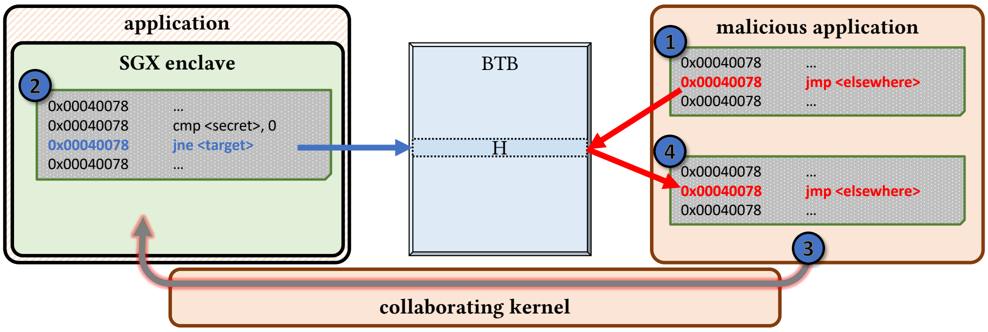


Figure 4.2. $\mathscr{A}$ can infer enclave branching behavior dependent on confidential data by observing how it affects the BPU history. The branch history H is first primed with a known state (❶), then the attacked enclave is allowed to execute (❷). The enclave execution will, depending on the confidential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

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, $\mathscr{A}$ first analyzes the enclave code and identifies a branch instruction to target. Although the enclave is in encrypted memory, $\mathscr{A}$ can observe the loading process and thus knowns the memory layout of the code sections. $\mathscr{A}$ 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, $\mathscr{A}$ primes H to a known state by repeatedly executing the shadow branch. $\mathscr{A}$ then allows the enclave to briefly execute before interrupting it again. After enabling the LBR performance counter, $\mathscr{A}$ executes the shadow branch again. If a misprediction is reported in the LBR, $\mathscr{A}$ 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 $\mathscr{A}$'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 (`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 $\mathscr{A}$ cannot 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 $\mathscr{A}$ 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 $\mathscr{A}$. Because the trampoline locations are randomized, $\mathscr{A}$ is forced to guess the location of the trampoline to shadow. Furthermore, because the BPU history is of limited size, $\mathscr{A}$ only has a limited number of guesses. With enough entropy, $\mathscr{A}$ 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, $\mathscr{A}$ is not limited to using one attack. For instance, by monitoring memory accesses, $\mathscr{A}$ 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, $\mathscr{A}$ 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, $\mathscr{A}$ 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, $\mathscr{A}$ 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.

5. ARM Pointer authentication

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. `W⊕X` policies 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 $\mathscr{A}$ 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-

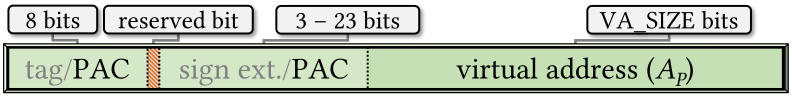


Figure 5.1. PA avoids both metadata and changing pointer size by embedding the PAC in the sign-extension bits of a pointer.

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

[1] ARM does not mandate the use of QARMA, but mentions it as an alternative [8].

[2] On 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

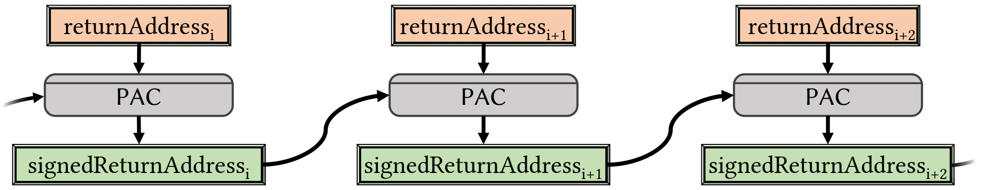


Figure 5.2. PACStack uses a PA to generate a chain of PACs that uniquely identifies a specific program execution path. Only the topmost `signedReturnAddress` is secured in a register to allow precise verification of return addresses.

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, $aret_i$, defined as:

$$\texttt{signedReturnAddress}_i = \text{sign}(\texttt{returnAddress}_i, \texttt{signedReturnAddress}_{i-1})$$

The previous signed return address, $\texttt{signedReturnAddress}_{i-1}$ is stored on the stack. The current `signedReturnAddress` is always kept in a register. To support this, the calling convention is changed such that the `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 `signedReturnAddress`. This allows functional compatibility with uninstrumented code, but still weakens the security guarantees if uninstrumented code is called.

Because the intermediate `signedReturnAddress` values are exposed in memory, $\mathscr{A}$ 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 $\mathscr{A}$ 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 `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 effectively 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 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-compete 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 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

[1] "Bulba" and "Kil3r". “Bypassing StackGuard and StackShield”. In: *Phrack* 10.56 (2000). URL: `http://phrack.org/issues/56/5.html` (visited on 11/19/2019) (cit. on p. 25).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-Flow Integrity”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. Alexandria, VA, USA, 2005, p. 340 (cit. on pp. 16, 27, 51, 56).

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-Flow Integrity Principles, Implementations, and Applications”. In: *ACM Transactions on Information and System Security*. TISSEC 13.1 (2009), 4:1–4:40 (cit. on pp. 27, 56).

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’07. Singapore, Republic of Singapore, 2007, pp. 312–320 (cit. on pp. 43, 45).

[5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. “Baggy Bounds Checking: An Ef Fi Cient and Backwards-Compatib Le Defense against Out-of-Bounds Errors”. In: *Proceedings of the 18th USENIX Security Symposium*. USENIX Security ’05 (2009), pp. 51–66 (cit. on p. 31).

[6] Lars Ole Andersen. “Program Analysis and Specialization for the C Programming Language”. Ph.D. Dissertation. DIKU, University of Copenhagen, 1994 (cit. on p. 23).

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. “Understanding the Mirai Botnet”. In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security ’17. Vancouver, BC, USA, 2017, pp. 1093–1110 (cit. on p. 15).

[8] ARM. *Architecture Reference Manual ARMv8*. DDI 0487C.a. 2017 (cit. on pp. 20, 32, 51, 52, 55).

[9] ARM. *Armv8.5-A Memory Tagging Extension*. Whitepaper. 2019 (cit. on p. 17).

[10] ARM Ltd. *Procedure Call Standard for the ARM 64-Bit Architecture*. ARM IHI 0055B. 2013 (cit. on p. 25).

[11] Abhishek Arya and Cris Neckar. *Fuzzing for Security — Chromium Blog*. 2012. URL: https://blog.chromium.org/2012/04/fuzzing-for-security.html (visited on 11/14/2019) (cit. on pp. 30, 59).

[12] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case for RISC-V*. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014 (cit. on p. 15).

[13] Roberto Avanzi. “The QARMA Block Cipher Family. Almost MDS Matrices over Rings with Zero Divisors, Nearly Symmetric Even-Mansour Constructions with Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes”. In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017), pp. 4–44 (cit. on p. 52).

[14] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. “The Meaning of Memory Safety”. In: *Proceedings of the International Conference on Principles of Security and Trust*. Ed. by Lujo Bauer and Ralf Küsters. Thessaloniki, Greece, 2018, pp. 79–105 (cit. on p. 29).

[15] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. “Conditional Model Checking: A Technique to Pass Information between Verifiers”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, NC, USA, 2012, 57:1–57:11 (cit. on p. 62).

[16] D. L. Bird and C. U. Munoz. “Automatic Generation of Random Self-Checking Test Cases”. In: *IBM Systems Journal* 22.3 (1983), pp. 229–245 (cit. on pp. 39, 59).

[17] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A Static Analyzer for Large Safety-Critical Software”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’03. San Diego, CA, USA, 2003, pp. 196–207 (cit. on p. 60).

[18] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. “Jump-Oriented Programming: A New Class of Code-Reuse Attack”. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. Hong Kong, China, 2011, pp. 30–40 (cit. on p. 26).

[19] Erik Bosman and Herbert Bos. “Framing Signals — A Return to Portable Shellcode”. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP ’14. San Jose, CA, USA, 2014, pp. 243–258 (cit. on p. 26).

[20] Rodrigo Rubira Branco. *PAX_REFCOUNT Documentation*. 2015. URL: https://forums.grsecurity.net/viewtopic.php?f=7&t=4173 (visited on 10/07/2019) (cit. on p. 38).

[21] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. “DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization”. In: arXiv (2019). arXiv: 1709.09917 [cs] (cit. on p. 44).

[22] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *Proceedings of the 11th USENIX Workshop on Offensive Technologies*. USENIX WOOT ’17. Vancouver, BC, 2017 (cit. on p. 44).

[23] Sergey Bratus, Michael Locasto, Meredith Patterson, Len Sassaman, and Anna Shubina. “Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation”. In: *USENIX ;login:* 36.6 (2011) (cit. on p. 62).

[24] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security ’17. Vancouver, BC, 2017, pp. 1041–1056 (cit. on p. 45).

[25] N. Burow, X. Zhang, and M. Payer. “SoK: Shining Light on Shadow Stacks”. In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP ’19. Los Alamitos, CA, USA, 2019, pp. 985–999 (cit. on pp. 17, 27, 57).

[26] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Computing Surveys* 50.1 (2017) (cit. on p. 56).

[27] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security ’19. Santa Clara, CA, USA, 2019, pp. 249–266 (cit. on pp. 16, 47).

[28] Javier Martinez Canillas. *Kbuild: The Linux Kernel Build System | Linux Journal*. 2012. URL: `https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system` (visited on 11/20/2019) (cit. on p. 34).

[29] Nicolas Carlini, Antonio Barresi, E T H Zürich, Mathias Payer, David Wagner, Thomas R Gross, E T H Zürich, Nicolas Carlini, Antonio Barresi, David Wagner, and Thomas R Gross. “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity”. In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security ’15. Washington, DC, USA, 2015, pp. 161–176 (cit. on pp. 27, 56, 62).

[30] Scott A. Carr and Mathias Payer. “DataShield: Configurable Data Confidentiality and Integrity”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIACCS ’17. Abu Dhabi, United Arab Emirates, 2017, pp. 193–204 (cit. on p. 40).

[31] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-Oriented Programming without Returns”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, IL, USA, 2010, p. 559 (cit. on p. 26).

[32] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution". In: *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*. EuroSP '19. Stockholm, Sweden, 2019, pp. 142–157 (cit. on p. 48).

[33] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th USENIX Security Symposium*. USENIX Security '05. Baltimore, MD, USA, 2005, pp. 177–191 (cit. on pp. 28, 51).

[34] Clang team. *AddressSanitizer — Clang 9 Documentation*. 2019. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/AddressSanitizer.html` (visited on 11/14/2019) (cit. on p. 30).

[35] Clang team. *Clang Command Line Argument Reference — Clang 9 Documentation*. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/ClangCommandLineReference.html` (visited on 11/14/2019) (cit. on pp. 25, 52, 60).

[36] Clang team. *Clang Static Analyzer — Clang 9 Documentation*. 2019. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/ClangStaticAnalyzer.html` (visited on 11/14/2019) (cit. on pp. 22, 60).

[37] Clang team. *Control Flow Integrity — Clang 9 Documentation*. 2019. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/ControlFlowIntegrity.html` (visited on 11/14/2019) (cit. on pp. 28, 59).

[38] Clang team. *ShadowCallStack — Clang 9 Documentation*. 2019. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/ShadowCallStack.html` (visited on 11/14/2019) (cit. on p. 28).

[39] George E. Collins. "A Method for Overlapping and Erasure of Lists". In: *Communications of the ACM* 3.12 (1960), pp. 655–657 (cit. on pp. 30, 34).

[40] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, CO, USA, 2015, pp. 952–963 (cit. on p. 27).

[41] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. "Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors". In: *Static Analysis*. Ed. by María Alpuente and Germán Vidal. Vol. 5079. Berlin, Heidelberg, 2008, pp. 62–77 (cit. on p. 62).

[42] Kees Cook. "Linux Kernel ASLR". In: *Linux Security Summit*. New Orleans, LA, USA, 2013 (cit. on pp. 26, 34).

[43] Kees Cook. *Security Things in Linux v4.11 — Codeblog*. 2017. URL: `https://outflux.net/blog/archives/2017/05/02/security-things-in-linux-v4-11/` (visited on 11/20/2019) (cit. on p. 19).

[44] Jonathan Corbet. *Kernel Building with GCC Plugins [LWN.Net]*. 2016. URL: `https://lwn.net/Articles/691102/` (visited on 11/20/2019) (cit. on pp. 34, 39).

[45] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016/086. 2016 (cit. on pp. 40, 43).

[46] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. "Protecting Systems from Stack Smashing Attacks with StackGuard". In: *Linux Expo*. 1999 (cit. on p. 25).

[47] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *Proceedings of the 7th USENIX Security Symposium*. Vol. 98. USENIX Security '98. San Antonio, TX, USA, 1998, pp. 63–78 (cit. on pp. 16, 25, 51, 53, 54).

[48] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. "The Performance Cost of Shadow Stacks and Stack Canaries". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Singapore, Republic of Singapore, 2015, pp. 555–566 (cit. on p. 27).

[49] Lucas Vincenzo Davi. "Code-Reuse Attacks and Defenses". Ph.D. Disseration. Technische Universität Darmstadt, 2015 (cit. on p. 26).

[50] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. "HAFIX: Hardware-Assisted Flow Integrity Extension". In: *Proceedings of the 52nd ACM/IEEE Annual Design Automation Conference*. DAC '15. San Francisco, CA, USA, 2015 (cit. on p. 31).

[51] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '14. San Diego, CA, USA, 2014, pp. 401–416 (cit. on pp. 27, 56, 62).

[52] Brooks Davis, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Robert N. M. Watson, Stacey Son, Jonathan Woodruff, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, and Nathaniel Wesley Filardo. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment". In: *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA, 2019, pp. 379–393 (cit. on p. 31).

[53] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In: *Proceedings of the 54th ACM/IEEE Annual Design Automation Conference*. DAC '17. Austin, TX, USA, 2017 (cit. on p. 57).

[54] Joe Devietti, Colin Blundell, M.M.K. Martin, and Steve Zdancewic. "HardBound: Architectural Support for Spatial Safety of the C Programming Language". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '08. Seatle, WA, USA, 2008, pp. 103–114 (cit. on pp. 17, 31).

[55] Will Dietz, Peng Li, John Regehr, and Vikram Adve. "Understanding Integer Overflow in C/C++". In: *ACM Transactions on Software Engineering and Methodology*. TOSEM 25.1 (2015), 2:1–2:29 (cit. on pp. 35, 38).

[56] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. "Efficient Protection of Path-Sensitive Control Security". In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security '17. Vancouver, BC, 2017, pp. 131–148 (cit. on p. 27).

[57] Stephen Dolan. "Mov Is Turing-Complete". In: *Computer Laboratory, University of Cambridge* (2013) (cit. on p. 47).

[58] Nurit Dor, Michael Rodeh, and Mooly Sagiv. "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, CA, USA, 2003, pp. 155–167 (cit. on p. 60).

[59] Thomas F. Dullien. "Weird Machines, Exploitability, and Provable Unexploitability". In: *IEEE Transactions on Emerging Topics in Computing* (2018) (cit. on p. 62).

[60] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. "Checked C: Making C Safe by Extension". In: *Proceedings of the 2018 IEEE Cybersecurity Development*. SecDev '18. Cambridge, MA, USA, 2018, pp. 53–60 (cit. on pp. 17, 30).

[61] Hiroaki Etoh and Kunikazu Yoda. *Protecting from Stack-Smashing Attacks*. IBM Research Division, Tokyo Research Laboratory, 2000 (cit. on p. 25).

[62] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '16. Taipei, Tawian, 2016, 40:1–40:13 (cit. on pp. 26, 45, 47).

[63] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA, 2018, pp. 693–707 (cit. on p. 45).

[64] Justin E Forrester and Barton P Miller. "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing". In: *Proceedings of the 4th USENIX Windows System Symposium*. Vol. 4. USENIX WinSys '00. Seattle. Seatle, WA, USA, 2000, pp. 59–68 (cit. on pp. 39, 59).

[65] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-Centric Defense Mechanism against Spectre Attacks". In: *Proceedings of the 56th ACM/IEEE Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA, 2019 (cit. on p. 48).

[66] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. "Buffer Overrun Detection Using Linear Programming and Static Analysis". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington DC, USA, 2003, pp. 345–354 (cit. on p. 60).

[67] GCC Wiki. *Intel® Memory Protection Extensions (Intel® MPX) Support in the GCC Compiler*. 2018. URL: `https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler` (visited on 10/07/2019) (cit. on pp. 36, 40).

[68] Xinyang Ge, Weidong Cui, and Trent Jaeger. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace". In: *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China, 2017, pp. 585–598 (cit. on p. 27).

[69] Ronald Gil, Hamed Okhravi, and Howard Shrobe. "There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection". In: *Proceedings of the 2018 IEEE Cybersecurity Development*. SecDev '18. Cambridge, MA, USA, 2018, pp. 102–109 (cit. on pp. 30, 36, 40).

[70] GNU. *GCC 9.2 Manual*. 2019. URL: `https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/` (visited on 11/10/2019) (cit. on pp. 25, 52).

[71] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing". In: *Proceedings of Hte 2008 Network and Distributed System Security Symposium*. Vol. 8. NDSS '08. San Diego, CA, USA, 2008, pp. 151–166 (cit. on p. 59).

[72] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. "Out of Control: Overcoming Control-Flow Integrity". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. San Jose, CA, USA, 2014, pp. 575–589 (cit. on pp. 27, 56, 62).

[73] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU". In: (March 2017) (cit. on p. 26).

[74] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR Is Dead: Long Live KASLR". In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Bonn, Germany, 2017, pp. 161–176 (cit. on p. 40).

[75] Reed Hastings and Bob Joyce. "Purify: Errors of Memory Leaks and Access Fast Detection". In: *Winter USENIX Conference*. 1992, pp. 125–136 (cit. on p. 30).

[76] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. "Dynamic Canary Randomization for Improved Software Security". In: *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. CISRC '16. Oak Ridge, TN, USA, 2016, 9:1–9:7 (cit. on pp. 25, 54).

[77] Michael Hicks. *What Is Memory Safety?* 2014. URL: `http://www.pl-enthusiast.net/2014/07/21/memory-safety/` (visited on 09/24/2019) (cit. on p. 29).

[78] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. "Enforcing Unique Code Target Property for Control-Flow Integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada, 2018, pp. 1470–1486 (cit. on p. 28).

[79] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. SP '16. San Jose, CA, USA, 2016, pp. 969–986 (cit. on pp. 28, 51, 62).

[80] Ralf Hund, Carsten Willems, Thorsten Holz, and Ruhr-university Bochum. "Practical Timing Side Channel Attacks Against Kernel Space ASLR". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. San Francisco, CA, USA, 2013 (cit. on p. 26).

[81] IBM. *IBM Cloud Data Shield*. 2019. URL: `https://www.ibm.com/cloud/data-shield` (visited on 10/11/2019) (cit. on p. 43).

[82] Intel. *Control-Flow Enforcement Technology Preview*. 2016 (cit. on pp. 17, 31, 53).

[83] Intel. *Intel 64 and Ia-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. 2019. URL: `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4` (visited on 11/08/2019) (cit. on pp. 28, 45).

[84] Intel. *Intel® Analysis of Speculative Execution Side Channels*. 336983-001. 2018 (cit. on p. 48).

[85] ISO/IEC. *Information Technology - Programming Languages - C*. INCITS/ISO/IEC 9899-2012. 2012 (cit. on p. 35).

[86] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. "Cyclone: A Safe Dialect of C". In: *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX ATC '02. Monterey, CA, USA, 2002, pp. 275–288 (cit. on p. 30).

[87] Ken Johnson and Matt Miller. "Exploit Mitigation Improvements in Windows 8". In: *Black hat USA* (2012) (cit. on p. 26).

[88] Simon Johnson. *Intel® SGX and Side-Channels*. 2018. URL: `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels` (visited on 11/20/2019) (cit. on p. 43).

[89] Richard W M Jones and Paul H J Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs." In: *Proceedings of the 3rd International Workshop on Automatic Debugging*. AADEBUG '97 1.1 (1997), pp. 13–26 (cit. on p. 31).

[90] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of the ACM on Programming Languages*. POPL '17 2 (POPL 2017), 66:1–66:34 (cit. on p. 61).

[91] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". In: *Proceedings of the 56th ACM/IEEE Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA, 2019 (cit. on p. 48).

[92] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software”. In: *Proceedings of the 2006 22nd Annual Computer Security Applications Conference*. ACSAC ’06. Miami Beach, FL, USA, 2006, pp. 339–348 (cit. on p. 26).

[93] Steve Klabnik and Carol Nichols. “Foreign Function Interface”. In: *The Rust Programming Language*. 2019 (cit. on p. 61).

[94] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP ’19. Los Alamitos, CA, USA, 2019 (cit. on p. 47).

[95] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology*. Ed. by Michael Wiener. CRYPTO ’99. Berlin, Heidelberg, 1999, pp. 388–397 (cit. on p. 44).

[96] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. “No Need to Hide: Protecting Safe Regions on Commodity Hardware”. In: *Proceedings of the 12th European Conference on Computer Systems*. EuroSys ’17. Belgrade, Serbia, 2017, pp. 437–452 (cit. on pp. 40, 61, 62).

[97] Tim Kornau. “Return Oriented Programming for the ARM Architecture”. Diplomarbeit. Bochum, Germany: Ruhr-Universität Bochum, 2009. URL: `http://bxi.es/Reversing-Exploiting/ROP/Return%20Oriented%20Programming%20for%20ARM.pdf` (cit. on pp. 26, 52).

[98] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “Spectre Returns! Speculation Attacks Using the Return Stack Buffer”. In: *Proceedings of the 12th USENIX Workshop on Offensive Technologies*. USENIX WOOT ’18. Baltimore, MD, USA, 2018 (cit. on p. 48).

[99] G Kroah-Hartman. “Kobjects and Krefs”. In: *Proceedings of the Linux Symposium*. Ottawa, ON, Canada, 2004, pp. 297–302 (cit. on p. 35).

[100] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. “SGXBOUNDS: Memory Safety for Shielded Execution”. In: *Proceedings of the 12th European Conference on Computer Systems*. EuroSys ’17. Belgrade, Serbia, 2017, pp. 205–221 (cit. on pp. 40, 41).

[101] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. “Code-Pointer Integrity”. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX OSDI ’14. Broomfield, CO, USA, 2014, pp. 147–163 (cit. on pp. 28, 51).

[102] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, and Dawn Song. “Poster: Getting the Point (Er): On the Feasibility of Attacks on Code-Pointer Integrity”. In: *IEEE Symposium on Security and Privacy*. 2015 (cit. on p. 28).

[103] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. "Sandcrust: Automatic Sandboxing of Unsafe Components in Rust". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS'17. Shanghai, China, 2017, pp. 51–57 (cit. on p. 61).

[104] William Landi. "Undecidability of Static Analysis". In: *ACM Letters on Programming Languages and Systems*. LOPLAS 1.4 (1992), pp. 323–337 (cit. on p. 60).

[105] Michael Larabel. *The New Features Of LLVM 9.0 & Clang 9.0 - Includes Building The Linux X86_64 Kernel - Phoronix*. 2019. URL: https://www.phoronix.com/scan.php?page=news_item&px=LLVM-9.0-Clang-9.0-Features (visited on 11/20/2019) (cit. on p. 34).

[106] Chris Lattner. "LLVM". In: *The Architecture of Open Source Applications*. Ed. by Amy Brown and Greg Wilson. 2014 (cit. on p. 22).

[107] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-Grained Control Flow inside SGX Enclaves with Branch Shadowing". In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security '17. Vancouver, BC, 2017, pp. 557–574 (cit. on pp. 45–47).

[108] Elias (Aleph One) Levy. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.19 (1996), p. 32 (cit. on pp. 16, 24).

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

[110] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security '19. Baltimore, MD, USA, 2018, pp. 973–990 (cit. on p. 47).

[111] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. "Transparent and Efficient CFI Enforcement with Intel Processor Prace". In: *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*. HPCA '17. 2017, pp. 529–540 (cit. on p. 27).

[112] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization". In: *Proceedings of Hte 2016 Network and Distributed System Security Symposium*. NDSS '16. San Diego, CA, USA, 2016 (cit. on pp. 27, 60).

[113] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. "CCFI: Cryptographically Enforced Control Flow Integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, CO, USA, 2015, pp. 941–951 (cit. on pp. 17, 28, 57).

[114] Nicholas D Matsakis and Felix S Klock II. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. Vol. 34. HILT '14. Portland, OR, USA, 2014, pp. 103–104 (cit. on pp. 16, 17, 22, 30, 60).

[115] Paul E Mckenney. *Overview of Linux-Kernel Reference Counting*. N2167=07-0027. IBM Beaverton: Linux Technology Center, 2007 (cit. on p. 35).

[116] Microsoft. *A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. 2006. URL: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in (visited on 09/05/2019) (cit. on pp. 16, 25, 51).

[117] Microsoft. *Azure Confidential Computing*. 2019. URL: https://azure.microsoft.com/en-us/solutions/confidential-compute/ (visited on 10/11/2019) (cit. on p. 43).

[118] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems*. Ed. by Wieland Fischer and Naofumi Homma. CHES '17. Taipei, Tawian, 2017, pp. 69–90 (cit. on p. 44).

[119] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. "Opaque Control-Flow Integrity". In: *Proceedings of the 2015 Network and Distributed System Security Symposium*. NDSS '15. San Diego, CA, USA, 2015 (cit. on p. 40).

[120] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. "DROP THE ROP Fine-Grained Control-Flow Integrity for the Linux Kernel". In: *Black Hat Asia* (2017) (cit. on p. 40).

[121] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland, 2009, pp. 245–258 (cit. on pp. 17, 31).

[122] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: Type-Safe Retrofitting of Legacy Software". In: *ACM Transactions on Programming Languages and Systems* 27.3 (2005), pp. 477–526 (cit. on p. 30).

[123] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi. "HardScope: Hardening Embedded Systems against Data-Oriented Attacks". In: *Proceedings of the 56th ACM/IEEE Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA, 2019 (cit. on pp. 17, 57).

[124] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis. Lecture Notes in Computer Science. 2017, pp. 259–284 (cit. on p. 31).

[125] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.2 (2019), 28:1–28:30 (cit. on pp. 17, 31, 36).

[126] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. “Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks”. In: *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX ATC ’18. Boston, MA, USA, 2018, pp. 227–240 (cit. on p. 44).

[127] Harish Patil and Charles Fischer. “Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs”. In: *Software: Practice and Experience*. SPE 27.1 (1997), pp. 87–110 (cit. on p. 31).

[128] PaX Team. *PaX Address Space Layout Randomization (ASLR)*. 2003. URL: `https://pax.grsecurity.net/docs/aslr.txt` (visited on 05/10/2018) (cit. on p. 26).

[129] PaX Team. *PaX PAGEEXEC Documentation*. 2006. URL: `https://pax.grsecurity.net/docs/pageexec.txt` (visited on 11/19/2019) (cit. on p. 25).

[130] Alexander (Solar Designer) Peslyak. *Getting around Non-Executable Stack (and Fix)*. 1997. URL: `https://seclists.org/bugtraq/1997/Aug/63` (visited on 09/05/2019) (cit. on pp. 16, 25, 51).

[131] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. “DynaGuard: Armoring Canary-Based Protections against Brute-Force Attacks”. In: *Proceedings of the 31st AnnualCcomputer Security Applications Conference*. ACSAC 2015. Los Angeles, CA, USA, 2015, pp. 351–360 (cit. on pp. 25, 54).

[132] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. “kRˆX: Comprehensive Kernel Protection against Just-in-Time Code Reuse”. In: *Proceedings of the 12th European Conference on Computer Systems*. EuroSys ’17. Belgrade, Serbia, 2017, pp. 420–436 (cit. on p. 40).

[133] Qualcomm. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. 2017 (cit. on pp. 17, 51, 52).

[134] Ramu Ramakesavan, Dan Zimmerman, Pavithra Singaravelu, George Kuan, Brian Vajda, Scott Gibbons, and Gautham Beeraka. *Intel® Memory Protection Extensions Enabling Guide (Rev 1.01)*. 2016 (cit. on p. 36).

[135] Gerardo Richarte et al. “Four Different Tricks to Bypass Stackshield and Stackguard Protection”. In: *World Wide Web* 1 (2002) (cit. on p. 25).

[136] Grigore Roşu, Wolfram Schulte, and Traian Florin ŞerbănuŢă. “Runtime Verification of C Memory Safety”. In: *Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Berlin, Heidelberg, 2009, pp. 132–151 (cit. on pp. 29, 60).

[137] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. “Evaluating the Effectiveness of Current Anti-Rop Defenses”. In: *Research in Attacks, Intrusions and Defenses*. Ed. by Angelos Stavrou, Herbert Bos, and Georgios Portokalidis. Rodney Bay, St. Lucia, France, 2014, pp. 88–108 (cit. on pp. 27, 56, 62).

[138] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs”. In: *Proceedings of the 2017 Network and Distributed System Security Symposium*. NDSS ’17. San Diego, CA, USA, 2017 (cit. on p. 47).

[139] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Presented as Part of the 2012 USENIX Annual Technical Conference*. USENIX ATC ’12. Boston, MA, USA, 2012, pp. 309–318 (cit. on p. 30).

[140] Kostya Serebryany. “ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety”. In: *USENIX ;login:* 44.2 (2019), p. 5 (cit. on p. 32).

[141] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. Vol. 22. CCS ’07. Alexandria, VA, USA, 2007, pp. 552–561 (cit. on pp. 24, 26, 51, 62).

[142] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. “On the Effectiveness of Address-Space Randomization”. In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS ’04. Washington DC, USA, 2004, pp. 298–307 (cit. on p. 16).

[143] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. In: *Proceedings of the 2017 Network and Distributed System Security Symposium*. NDSS ’17. San Diego, CA, USA, 2017 (cit. on pp. 44, 45).

[144] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. “Preventing Page Faults from Telling Your Secrets”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. Xi’an, China, 2016, pp. 317–328 (cit. on p. 45).

[145] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. “Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP ’13. San Francisco, CA, USA, 2013, pp. 574–588 (cit. on p. 27).

[146] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. “HDFI: Hardware-Assisted Data-Flow Isolation”. In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. SP ’16. San Jose, CA, USA, 2016 (cit. on p. 57).

[147] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. “SoK: Sanitizing for Security”. In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP ’19. Los Alamitos, CA, USA, 2019, pp. 1275–1295 (cit. on p. 59).

[148] Wei Song, Alex Bradbury, and Robert Mullins. “Towards General Purpose Tagged Memory”. In: *Proceedings of the 2nd RISC-V Workshop*. Berkeley, CA, USA, 2015 (cit. on p. 57).

[149] Eugene H. Spafford. “The Internet Worm Program: An Analysis”. In: *ACM SIGCOMM Computer Communication Review*. SIGCOMM 19.1 (1989), pp. 17–57 (cit. on pp. 16, 24).

[150] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, FL, USA, 1996, pp. 32–41 (cit. on p. 23).

[151] László Szekeres, Mathias Payer, and Tao Wei. “SoK: Eternal War in Memory”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP ’13. San Francisco, CA, USA, 2013, pp. 48–62 (cit. on pp. 16, 29).

[152] the Kernel development community. *Coccinelle*. 2019. URL: `https://www.kernel.org/doc/html/v4.15/dev-tools/coccinelle.html` (visited on 10/08/2019) (cit. on p. 38).

[153] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM”. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security ’14. San Diego, CA, USA, 2014, pp. 941–955 (cit. on pp. 27, 28).

[154] Stylianos Tsampas et al. “Towards Automatic Compartmentalization of C Programs on Capability Machines”. In: *Workshop on Foundations of Computer Security 2017*. 2017 (cit. on pp. 61, 62).

[155] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. “ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK)”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security ’19. Santa Clara, CA, USA, 2019, pp. 1221–1238 (cit. on pp. 31, 61, 62).

[156] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, Raoul Strackx, and Ku Leuven. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security ’18. Baltimore, MD, USA, 2018, pp. 991–1008 (cit. on p. 48).

[157] Jo Van Bulck, Frank Piessens, Raoul Strackx, Jo Van Bulck, K U Leuven, Frank Piessens, K U Leuven, Raoul Strackx, Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*. SysTEX’17. Shanghai, China, 2017, 4:1–4:6 (cit. on pp. 46, 47).

[158] J. Vanegue. “The Weird Machines in Proof-Carrying Code”. In: *Proceedings of the 2014 IEEE Security and Privacy Workshops*. SPW ’14. 2014, pp. 209–213 (cit. on p. 62).

[159] Wei Wang, Clark Barrett, and Thomas Wies. “Partitioned Memory Models for Program Analysis”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ahmed Bouajjani and David Monniaux. Paris, France, 2017, pp. 539–558 (cit. on p. 62).

[160] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. “To Detect Stack Buffer Overflow with Polymorphic Canaries”. In: *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DNS ’18. Luxembourg City, 2018, pp. 243–254 (cit. on pp. 25, 54).

[161] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. “Shuffler: Fast and Deployable Continuous Code Re-Randomization”. In: *Proceedings of the 12th USENIX Symposium on Pperating Systems Design and Implementation*. USENIX OSDI ’16. Savannah, GA, USA, 2016, pp. 367–382 (cit. on p. 27).

[162] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. In: *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*. ISCA ’14. 2014, pp. 457–468 (cit. on pp. 17, 31).

[163] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. “Transparent Runtime Randomization for Security”. In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. 2003, pp. 260–269 (cit. on p. 26).

[164] Wei Xu, Daniel C. DuVarney, and R. Sekar. “An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs”. In: *Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering*. SIGSOFT ’04/FSE-12. Newport Beach, CA, USA, 2004, pp. 117–126 (cit. on p. 31).

[165] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. “CONFIRM: Evaluating Compatibility and Relevance of Control-Flow Integrity Protections for Modern Software”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security ’19. Santa Clara, CA, USA, 2019, pp. 1805–1821 (cit. on pp. 56, 59).

[166] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP ’14. San Jose, CA, USA, 2015, pp. 640–656 (cit. on p. 45).

[167] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security ’14. San Diego, CA, USA, 2014, pp. 719–732 (cit. on pp. 16, 21, 44, 48).

[168] Yves Younan, Wouter Joosen, Frank Piessens, and Report Cw. *Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures*. CW 386. 2004, p. 82 (cit. on pp. 29, 30, 35, 51).

[169] Mohamed A. El-Zawawy. “Novel Designs for Memory Checkers Using Semantics and Digital Sequential Circuits”. In: *Computational Science and Its Applications*. Ed. by Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Marina L. Gavrilova, Ana Maria Alves Coutinho Rocha, Carmelo Torre, David Taniar, and Bernady O. Apduhan. ICCSA ’15. Salvador, Bahia, Brazil, 2015, pp. 597–611 (cit. on pp. 29, 60).

[170] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. “ARMlock: Hardware-Based Fault Isolation for ARM”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA, 2014, pp. 558–569 (cit. on p. 62).

[171] Peter Zijlstra. *Re: [Tip:Locking/Core] Refcount_t: Introduce a Special Purpose Refcount Type*. 2017. URL: `https://patchwork.kernel.org/patch/9569859/` (visited on 11/06/2019) (cit. on p. 38).

Errata

Publication III

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

Publication I

Elena Reshetova, Hans Liljestrand, Andrew Paverd, N. Asokan. Towards Linux Kernel Memory Safety. *Software: Practice and Experience*, December 2018.

© 2018 John Wiley & Sons, Ltd.
Reprinted with permission.

Received: 3 April 2018 | Revised: 17 July 2018 | Accepted: 9 August 2018
DOI: 10.1002/spe.2638

RESEARCH ARTICLE

WILEY

Toward Linux kernel memory safety

Elena Reshetova[1] | Hans Liljestrand[2] | Andrew Paverd[2] | N. Asokan[2]

[1]Intel OTC, Espoo, Finland

[2]Aalto University, Espoo, Finland

Correspondence
Elena Reshetova, Intel OTC, 02160 Espoo, Finland.
Email: elena.reshetova@intel.com

Funding information
Intel Collaborative Research Institute for Secure Computing at Aalto University; Cloud Security Services (CloSer) project, funded by Tekes/Business Finland, Grant/Award Number: 3881/31/2016

Summary

The security of billions of devices worldwide depends on the security and robustness of the mainline Linux kernel. However, the increasing number of kernel-specific vulnerabilities, especially memory safety vulnerabilities, shows that the kernel is a popular and practically exploitable target. Two major causes of memory safety vulnerabilities are reference counter overflows (temporal memory errors) and lack of pointer bounds checking (spatial memory errors). To succeed in practice, security mechanisms for critical systems like the Linux kernel must also consider performance and deployability as critical design objectives. We present and systematically analyze two such mechanisms for improving memory safety in the Linux kernel, ie, (1) an overflow-resistant reference counter data structure designed to securely accommodate typical reference counter usage in kernel source code and (2) runtime pointer bounds checking using Intel memory protection extension in the kernel. We have implemented both mechanisms and we analyze their security, performance, and deployability. We also reflect on our experience of engaging with Linux kernel developers and successfully integrating the new reference counter data structure into the mainline Linux kernel.

KEYWORDS

Linux kernel, Linux kernel development process, memory safety

1 | INTRODUCTION

The Linux kernel forms the foundation of billions of different devices, ranging from servers and desktops to smartphones and embedded devices. There are many solutions for strengthening Linux application security, including access control frameworks (SELinux,[1] AppArmor[2]), integrity protection systems (integrity measurement architecture/earned value management,[3] dm-verity[4]), encryption, key management, and auditing. However, these are all rendered ineffective if an attacker gains control of the kernel. Recent trends in common vulnerabilities and exposures (CVEs) indicate a renewed interest in kernel vulnerabilities.[5] On average, it takes 5 years for a kernel bug to be found and fixed,[6] and even when fixed, security updates might not be deployed to all vulnerable devices. Therefore, we cannot rely solely on retroactive bug fixes, but require *proactive* measures to harden the kernel against potential vulnerabilities. This is the goal of the Kernel Self

© 2018 John Wiley & Sons, Ltd.

Protection Project,[7] a large community of volunteers working on the mainline Linux kernel. In this paper, we describe our contributions, as part of the Kernel Self Protection Project, to the development of two kernel memory safety mechanisms.

Depending on their severity, memory errors can allow an attacker to read, write, or execute memory, thus making them attractive targets. For example, *use-after-free* errors and *buffer overflows* feature prominently in recent Linux kernel CVEs.[8,9] Memory errors arise due to the lack of inherent memory safety in C, the main implementation language of the Linux kernel, and can be divided into two fundamental classes.

Temporal memory errors occur when pointers to freed/uninitialized memory are dereferenced. For example, a *use-after-free* error occurs when dereferencing a pointer that has been prematurely freed by another execution thread. The Linux kernel is vulnerable to temporal memory errors because it is written in C, and thus does not benefit from automated garbage collection. Instead, kernel object lifetimes are managed using *reference counters*.[10] Whenever a new reference to an object is taken, the object's reference counter is incremented, and whenever the object is released, the counter is decremented. When the counter reaches zero, the object can be safely destroyed and its memory freed. Reference counters in the Linux kernel are typically implemented using the `atomic_t` type,[11] which is, in turn, implemented as an `int` with a general purpose atomic API consisting of over 100 functions. This can give rise to temporal memory errors since `atomic_t` can overflow, as was the case in, eg, CVE-2014-2851, CVE-2016-4558, and CVE-2016-0728.

Spatial memory errors occur when pointers are used to access memory outside the bounds of their intended areas. For example, a buffer overflow occurs when the amount of data written exceeds the size of the target buffer. The Linux kernel is vulnerable to spatial memory errors as any other piece of C code due to the bugs introduced by its developers. Spatial memory errors in the mainline Linux kernel are pretty common and have appeared in, eg, CVE-2014-0196, CVE-2016-8440, CVE-2016-8459, and CVE-2017-7895.

Although memory safety has been scrutinized for decades (Section 3), much of the work has focused on user space. These solutions are not readily transferable to kernel space. For example, schemes like SoftBound[12] defend against spatial memory errors by storing pointer metadata in a disjoint data structure using a fast static addressing scheme (eg, a large hash table). However, the range of memory addresses required for this data structure far exceeds the physical memory available on most systems. This is only possible for user space applications because the kernel can handle page faults when a virtual address has not yet been mapped to physical memory. Since the kernel cannot handle its own page faults, this type of scheme cannot be used in kernel space.

Although there have been a small number of proposals for improving kernel memory safety (eg, kCFI[13] and KENALI[14]), these have not considered the critical issue of deployability in the mainline Linux kernel. Other mechanisms, such as the widely used kernel address sanitizer (KASAN),[15] are intended as debugging facilities, not runtime protection.

Contributions: In this paper, we present a solution for mitigating one of the major causes of temporal errors, and a solution for mitigating spatial memory errors in general. Specifically, we claim the following contributions.

- **Extended `refcount_t` API**: After performing a kernel-wide analysis of reference counters, we contributed to the design of `refcount_t`, a new reference counter data type that prevents reference counter overflows and significantly reduces the complexity of the previous reference counter design (Section 5.3).
- **Converting reference counters**: Locating reference counters in the kernel is nontrivial due to the diverse implementations and kernel-wide distribution. To achieve this, we developed a heuristic technique to identify instances of reference counters in the kernel source code (Section 5.1). Using this technique, we converted the core parts of the kernel source tree to use the new `refcount_t` API (Section 5.6). At the time of writing, 170 of our 233 `refcount_t` conversion patches have already been integrated.
- **Memory protection extension for kernel (MPXK)**: We present a new spatial memory error prevention mechanism for the Linux kernel based on the recently released Intel memory protection extensions (MPXs) (Section 6). Applying MPX to the kernel is nontrivial due the strict performance and memory requirements. In particular, this required a complete redesign of how MPX memory is handled in the kernel, compared with the user space MPX variant.
- **Evaluation**: We present a systematic analysis of `refcount_t` and MPXK in terms of performance, security, and usability for kernel developers (Section 7). Such analysis is particularly challenging for nonfunctional features where microbenchmarks are not representative of real-world impact and target platforms are diverse. In the interest of reproducibility, we make source code and test setups available at https://github:com/ssg-kernel-memory-safety.
- **Experience**: One of our primary considerations was to develop memory protection techniques that can be deployed in the mainline Linux kernel. As demonstrated by our conversion of kernel reference counters, our efforts have been successful. In Section 8, we reflect on our experience of working with the Linux kernel maintainers and offer suggestions for other researchers with the same objective.

2 | BACKGROUND

2.1 | Linux kernel reference counters

Temporal memory errors typically arise in systems that do not have automated mechanisms for object destruction and memory deallocation (ie, garbage collection). In simple nonconcurrent C programs, objects typically have well defined and predictable allocation and release patterns, which make it trivial to free them manually. However, in complex systems like the Linux kernel, objects are extensively shared and reused to minimize CPU and memory use. For example, in the Linux kernel, everything from filesystem nodes to group permission data structures are shared and reused. To enable this sharing sharing and reuse of kernel objects, the Linux kernel makes extensive use of reference counters.[10] However, reference counting schemes are historically error prone; a missed increment or decrement, often in an obscure code path, could imbalance the counter and cause a use-after-free error.

As explained in Section 1, reference counters in the Linux kernel are typically implemented using the `atomic_t` type,[11] which is, in turn, implemented as an `int`, which can thus overflow. In other words, this type of reference counter can be reset to zero using only increments, which will inevitably result in the object being prematurely freed, leading to a use-after-free error. Overflow bugs are particularly hard to detect using static code analysis or fuzzing techniques because they require many consequent iterations to trigger the overflow.[16] A recent surge of exploitable errors, such as CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2017-7487, and CVE-2017-8925, specifically target reference counters. In addition, the general purpose API also provides ample room for subtly incorrect reference counting schemes motivated by performance or implementation shortcuts.

2.2 | Intel MPXs

Intel MPXs[17] is a recent technology to prevent spatial memory errors. It is supported on both Pentium core and Atom core microarchitectures from Skylake and Goldmount onwards, thus targeting a significant range of end devices from desktops to mobile and embedded processors. In order to use the MPX hardware, an application must be compiled with an MPX-enabled compiler, such as GCC or Intel C++ Compiler (ICC), so that the resulting binary includes the new MPX instructions. In GNU Compiler Collection (GCC), this support is built around the architecture-independent *pointer bounds checker*. At present, these compilers only support MPX for user space applications. Furthermore, the application must be run on an MPX-enabled operating system, which manages the MPX metadata. Memory protection extension is source and binary compatible, meaning that no source code changes are required and that MPX-instrumented binaries can be linked with nonMPX binaries (eg, system libraries). The MPX-instrumented binaries can still be run on all systems, including legacy systems without MPX hardware. If MPX hardware is available, the instrumented binaries will automatically detect and configure this hardware when the application is initialized. However, the checks are performed by compiler instrumentation and can only cover compiler generated code, not, for instance, assembly.

Memory protection extension prevents spatial memory errors by checking pointer bounds before a pointer is dereferenced. Every pointer is associated with an upper and lower bound, which are determined by means of compile-time code instrumentation. For a given pointer, the bounds can either be set statically (eg, based on static data structure sizes), or dynamically (eg, using instrumented memory allocators).

The MPX instrumentation uses the new `bndcl` and `bndcu` instructions to explicitly check the bounds before any use of a pointer, so the type of access performed by the pointer (read, write, or execute) does not matter. For example, `malloc` is instrumented to set the bounds of newly allocated pointers. When pointers are derived from others, eg, a pointer into a substructure is taken, the bounds are typically *narrowed* to that of the substructure. The narrowed bounds then behave as other bounds, ie, they are checked upon pointer dereference. This narrowing is omitted in specific cases that indicate noncompatible use. For instance, a pointer to the first element of an array is often used as an iterator; hence, its bounds cannot be narrowed. Memory protection extension does consider pointer arithmetic and will not modify bounds based on such modifications.

In order to perform a bounds check, the pointer's bounds must be loaded in one of the four new MPX bound (`bndx`) registers. Memory protection extension stores pointer bounds separately from the pointer itself, either in a special purpose data structure, the stack, or in static memory. In some cases, the bounds to be stored cannot be determined at compile time, preventing the use of static or stack memory (eg, a dynamically sized array of pointers). In these cases, bounds are stored in memory in a new two-level metadata structure and accessed using the new bound load (`bndldx`) and store (`bndstx`) instructions. As shown in Figure 1, these instructions use bits 20 to 47 of the pointer's address that are used as

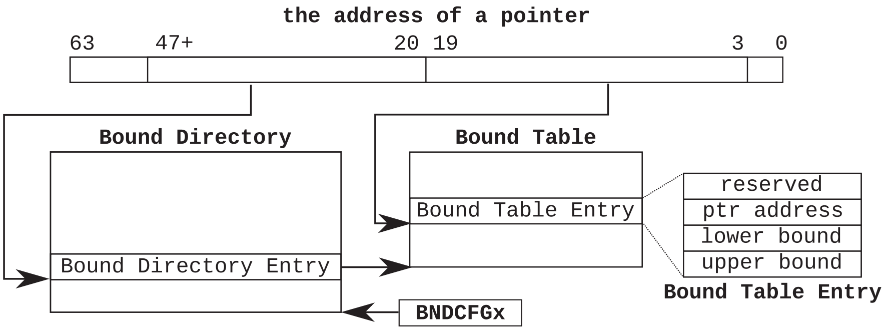


FIGURE 1 The memory protection extension bounds addressing scheme uses the 2 GB bound directory to lookup 4 MB bound tables, which, in turn, contain the bounds information for individual pointers (sizes reported for 64-bit architectures). Using virtual memory and paging, the kernel only fills/allocates these data structures on demand, thus keeping actual memory use relatively low

an index into the process's bound directory (BD), which contains pointers to the relevant bound tables (BTs). Bits 3 to 19 of the pointer's address are then used as an index into the specific BT that contains the address and bounds for the pointer. On 64-bit systems, the BD is 2 GB and each BT is 4 MB, which means that a BT can accommodate metadata for up to 2^{17} pointers. A pathological case with dedicated physical memory would thus require a base of 2 GB in addition to 4 MB for each pointer. To reduce this high memory overhead, the Linux kernel only allocates physical memory to the regions of the BD that are actually used, and only allocates individual BTs when they are accessed. When a bounds check fails, the CPU issues an exception that must be handled by the kernel. Current kernel implementations allow user space processes to register their own error handlers, which can then either log or abort on bound violations.

3 | RELATED WORK

Research on memory safety has long roots in both industry and academia, with many solutions proposed to detect, mitigate, or eliminate various types of memory errors. However, the vast majority of these have not treated deployability as a primary consideration, and so have not been deployed or used in real systems. Purify,[18] Shadow Guarding,[19,20] SoftBound,[12] and other similar approaches[21-25] have unacceptably high runtime performance overhead. CCured[26] and Cyclone[27] require source code changes and are not backward compatible.

Moreover, these solutions have focused on user space and usually cannot be applied in kernel space. Some recent notable exceptions are kCFI[13] and KENALI,[14] but from an implementation standpoint, neither of these have targeted integration with the mainline kernel. To our knowledge, none of these are used as runtime security mechanisms in production systems. A notable exception is the PaX/Grsecurity patches that have pioneered many in-use security mechanisms, such as address space layout randomization.[28] PaX/Grsecurity also includes a `PAX_REFCOUNT`[29] feature that prevents reference counter overflows, but requires extensive modification of the underlying atomic types. These extensive changes, and a potential race condition, made this feature unsuitable for mainline kernel adoption.[29]

Other tools are widely used to debug memory errors during development. For example, Valgrind[30] offers a suite of six tools for debugging, profiling, and error detection, including the detection of memory errors. AddressSanitizer[31] is arguably the state of the art in runtime memory error detection, but is unsuitable for production use due to performance overhead. KASAN[15] is integrated into the mainline Linux kernel and offers similar protections for the kernel, but again incurs performance overheads that are unacceptably high for most production use cases.

From a production perspective, much work has focused on preventing the exploitation of memory errors. Exploitability of buffer overflows, whether stack based or heap based, can be limited by preventing an attacker from misusing overflown data. One early mitigation technique is the nonexecutable bit for stack, heap, and data pages.[32,33] However, this can be circumvented by using overflows for execution redirection into other legitimate code memory, such as the C library in so called return-to-libc attacks, or more generally to any executable process memory in return-oriented programming attacks.[34] Another mitigation technique is the use stack canaries to detect stack overflows, eg, StackGuard[35] and StackShield.[36] However, these detection techniques can typically be circumvented using more selective overflows that avoid the canaries, or by exploiting other callback pointers such as exception handlers. Probabilistic mitigation techniques, such as memory randomization,[28,37] are commonly used, but have proven difficult to secure against indirect and direct memory leaks that divulge the randomization patterns, or techniques such as heap spraying.[38]

In contrast to the development/debugging temporal safety tools and the runtime friendly mitigation measures, MPXK is a runtime efficient system focused on the prevention of the underlying memory errors. The MPXK shares some conceptual similarities with previous solutions.[12,26,27,39,40] Like these systems, MPXK does not use fat pointers, which alter the implementation of pointers to store both the pointer and the bounds together, but instead preserves the original memory layout. Unlike purely software-based systems, such as SoftBound,[12] MPXK has the advantage of hardware registers for propagating bounds and hardware instructions for enforcing bounds. HardBound[41] employs hardware support similar to MPXK, but has a worst-case memory overhead of almost 200%. However, HardBound has only been simulated on the micro-operation level and lacks any existing hardware support. Unlike MPXK, none of these previous schemes have been designed for use in the kernel.

4 | PROBLEM STATEMENT

As explained previously, two major causes of memory errors are (1) temporal memory errors caused by reference counter overflows and (2) spatial memory errors arising from out-of-bounds memory accesses. Our objective is to develop solutions that *proactively* protect the Linux kernel against these two causes of memory errors. Specifically, we define the following requirements.

- **Reference counter overflows**: We require a reference counter type and associated API in which the counter is *guaranteed never to overflow*.
- **Out-of-bounds memory accesses**: We require an access control scheme for kernel memory that *prevents out-of-bounds accesses*.

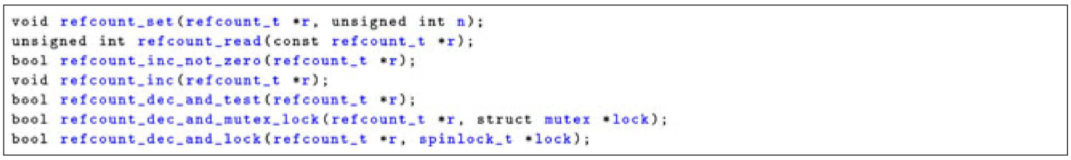
In addition to the aforementioned requirements, the following are mandatory design considerations in order for the aforementioned solutions to be used in practice.

- **Performance:** Some kernel subsystems, such as networking and filesystem, have strict performance requirements. Any security mechanisms that are perceived to add unacceptable performance penalty risk being rejected by the maintainers of these subsystems. However, there is no kernel-wide definition of what constitutes an unacceptable performance penalty, as this differs per subsystem. Furthermore, the Linux kernel runs on a vast range of devices, including closed fixed-function devices like routers, where software attacks are not a threat but performance requirements are stringent. Thus, the balance between security and performance must be configurable to meet the constraints of different usage scenarios.
- **Deployability:** In order to be integrated into the mainline Linux kernel, a solution should follow the Linux kernel design guidelines, minimize the number and extent of kernel-wide changes, and be sufficiently easy to use and maintain. *Usability for kernel developers* is particularly crucial for new features that may be adopted at the discretion of subsystem developers.

5 | REFERENCE COUNTER OVERFLOWS

A prerequisite for the analysis and conversion of reference counters is to identify all uses of reference counters in the kernel. This is nontrivial because (1) reference counters are often implemented using a general purpose atomic integer type, which is also used for other purposes; and (2) not all reference counters follow conventional practices. Unconventional reference counter implementations cannot be ignored because these are usually more likely to be error prone, and would thus benefit the most from our security mechanism.

Recently, a new `refcount_t` type and a corresponding minimal API were introduced by one of the Linux maintainers, Peter Zijlstra. This prevents incrementing a reference counter from zero or overflowing the counter. However, our kernel-wide analysis showed that the `refcount_t` API was too restrictive to be used in certain kernel subsystems. To overcome this, we proposed several additions to the initial `refcount_t` API, making it widely usable as a replacement for existing reference counters. Using this improved API, we have developed a set of patches to *convert all conventional reference counters in the kernel* to use `refcount_t`.



```
void refcount_set(refcount_t *r, unsigned int n);
unsigned int refcount_read(const refcount_t *r);
bool refcount_inc_not_zero(refcount_t *r);
void refcount_inc(refcount_t *r);
bool refcount_dec_and_test(refcount_t *r);
bool refcount_dec_and_mutex_lock(refcount_t *r, struct mutex *lock);
bool refcount_dec_and_lock(refcount_t *r, spinlock_t *lock);
```

Listing 1 Initial bare `refcount_t` API [Colour figure can be viewed at wileyonlinelibrary.com]

5.1 | Analyzing Linux kernel reference counters

We use Coccinelle,[42] a static analyzer integrated into the Linux kernel build system (KBuild), to systematically analyze the kernel source code and locate all reference counters based on the `atomic_t` type. Coccinelle takes code patterns as input and finds (or replaces) their occurrences in the given code base. We defined three such code patterns to identify reference counters based on their behavior (see Listing A1 in the Appendix for the full patterns).

1. Using `atomic_dec_and_test` (or one of its variants) to decrement an atomic variable, testing if the resulting value is zero, and if so, freeing a referenced object. This is the archetypical reference counter use case.
2. Using `atomic_add_return` to decrement a variable (by adding −1), and comparing its updated value against zero. This is a variation of the basic `dec_and_test` case using a different function.
3. Using `atomic_add_unless` to decrement a counter only when its value is not one. This case is less common.

These patterns are strong indicators that the identified object employs an `atomic_t` reference counting scheme. So far, this approach has detected all occurrences of reference counters. Some false positives were reported, particularly in implementations that make use of `atomic_t` variables for purposes other than reference counting. For example, under one condition, an object might be freed when the counter reaches zero (like a reference counter), but under a different condition, the object might instead be recycled when the counter reaches zero (see Section 5.5 for examples). Of the 250 `atomic_t` variables reported on an unmodified v4.10 kernel, we have manually confirmed 233 variables as reference counters.

5.2 | `refcount_t` API

The initial `refcount_t` API, introduced by Peter Zijlstra* (Listing 1), was designed around strict semantically correct reference counter use. This meant that beyond `set` and `read` calls, the API provided only two incrementing functions and three decrementing functions.

Both incrementing functions refuse to increment a counter when its value is zero to avoid potential use-after-free errors as illustrated in Figure 2. The `refcount_inc_not_zero` function returns a `true` value if the counter was nonzero, indicating that the referenced object is safe to use. Alternatively, `refcount_inc` can be used to avoid redundant checks in situations where the resulting value will definitely be positive.

The three decrementing functions return a value of `true` if the counter value reaches zero, and include compile-time warnings to ensure that this return value is checked. When a reference counter reaches zero, the object should always be released to avoid a memory leak. The `refcount_dec_and_mutex_lock` and `refcount_dec_and_lock` atomically combine the counter decrement with acquiring a mutex or lock.

The main challenge in designing the `refcount_t` API was how to deal with an event that would otherwise cause the counter to overflow. One approach would be to simply ignore the event, such that the counter remains at its maximum value. However, this means that the number of references to the object will be greater than the maximum counter value. If the counter is subsequently decremented, it will reach zero and the object will be freed before all references have been released, leading to a use-after-free error. To overcome this challenge, the `refcount_t` API instead *saturates* the counter, such that it remains at its maximum value, even if there are subsequent increment *or decrement* events (Algorithms 1 and 2). A saturated counter would, therefore, result in a memory leak since the object will never be freed. However, a cleanly logged memory leak is a small price to pay for avoiding the potential security vulnerabilities of a reference counter overflow. This approach is similar to that previously used by the PaX/Grsecurity patches.[29]

*http://lwn.net/Articles/713645/

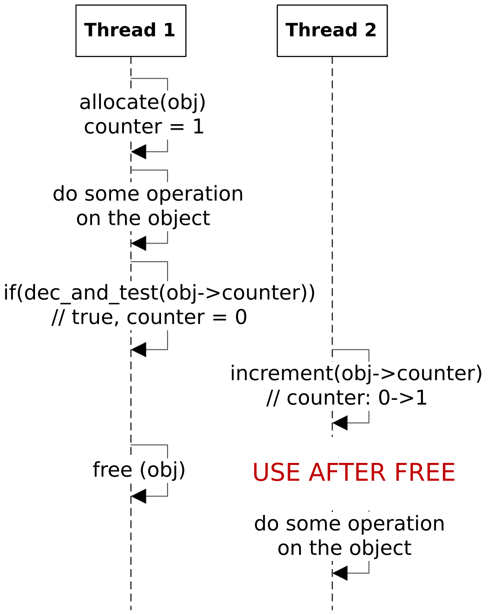


Figure 2 Potential use-after-free when incrementing a reference counter from zero [Colour figure can be viewed at wileyonlinelibrary.com]

Algorithm 1 `High-level refcount_inc_and_test` saturation behavior. Instead of allowing overflow, the counter is set to its maximum value, thus preventing any further changes

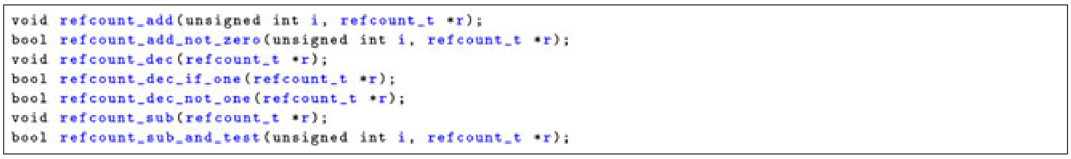
```
if $value == 0$ then
    return false
end if
if $value + 1 < value$ then
    $value \leftarrow$ MAX_VALUE                ▷ Saturate value on overflow
else
    $value \leftarrow value + 1$
end if
return true
```

Algorithm 2 `High-level refcount_dec_and_test` saturation behavior. A return value of true indicates that the value is non-zero and that the referenced object is usable, not whether the decrement actually took place or not. Note that underflows are prevented and treated exactly as a successful non-zero decrement

```
if $value ==$ MAX_VALUE then
    return false                              ▷ Value is saturated
end if
if $value - 1 > value$ then
    return false                              ▷ Underflow
end if
$value \leftarrow value - 1$
return $value == 0$                           ▷ Return value is true only when reaching zero
```

5.3 | Our extensions to the `refcount_t` API

We conducted a systematic analysis of existing reference counters in the Linux kernel source code. This revealed several variations of the strict archetypical reference counting schemes that are incompatible with the initial `refcount_t` API.



```
void refcount_add(unsigned int i, refcount_t *r);
bool refcount_add_not_zero(unsigned int i, refcount_t *r);
void refcount_dec(refcount_t *r);
bool refcount_dec_if_one(refcount_t *r);
bool refcount_dec_not_one(refcount_t *r);
void refcount_sub(refcount_t *r);
bool refcount_sub_and_test(unsigned int i, refcount_t *r);
```

Listing 2 Our additions to the `refcount_t` API [Colour figure can be viewed at wileyonlinelibrary.com]

As a result, we designed several API additions that have subsequently been integrated into the Linux kernel.[†] Our new API calls are shown in Listing 2.

The functions allowing arbitrary additions and subtractions, ie, `refcount_add` and `refcount_sub`, are necessary for situations in which larger value changes occur. For example, the `sk_wmem_alloc`[‡] variable in the networking subsystem serves as a reference counter but also tracks the transfer queue, and is thus subject to arbitrary additions and subtractions. The return values of `refcount_sub_and_test` and `refcount_add_not_zero` indicate whether the counter has reached zero. We introduced `refcount_dec` to accommodate situations in which the counter will definitely be nonzero and a forced return value check would incur needless overhead. For instance, some functions in the `btrfs` filesystem[§] handle nodes that are guaranteed to be cached, ie, there will always be at least one reference held by the cache. Finally, `refcount_dec_if_one` and `refcount_dec_not_one` enable schemes that require specific operations before or instead of releasing objects. For instance, the networking subsystem extensively uses patterns where a reference counter value of one indicates that an object is invalid but can be recycled.

5.4 | Implementation considerations

To avoid costly lock or mutex use, the generic implementation of the `refcount_t` API, ie, nonarchitecture-specific implementation, uses the *compare-and-swap* pattern, which is built around the atomic `atomic_cmpxchg` function shown in Algorithm 3. On x86, `atomic_cmpxchg` is implemented as a single atomic CPU instruction (`cmpxchg`), but the implementation is guaranteed to be atomic regardless of architecture. The function always returns the prior value, but exchanges it only if it was equal to the given condition value *comp*. Compare-and-swap works by indefinitely looping until a `cmpxchg` succeeds. This avoids costly locks and allows all but the `cmpxchg` to be nonatomic. Note that in the typical case, without concurrent modifications, the loop runs only once, thus being much more efficient than synchronization mechanisms (eg, a lock or mutex).

Algorithm 3 `cmpxchg`(*atomic*, *comp*, *new*) sets the value of *atomic* to *new* if, and only if, the prior value of *atomic* was equal to *comp*. Irrespective of whether the value was changed, the function always returns the prior value of *atomic*. In practice this function is implemented as a single inline CPU instruction

```
$old \leftarrow atomic.value$
if $old = comp$ then
    $atomic.value \leftarrow new$
end if
return $old$
```

As an example of the extended `refcount_t` API implementation, consider the `refcount_add_not_zero` function shown in Algorithm 4. It is used to increase `refcount_t` when acquiring a reference to the associated object, and a return value of `true` indicates that the counter was nonzero and thus the associated object is safe to use. The actual value from a user perspective is irrelevant and `refcount_add_not_zero` guarantees only that the return value is true if and only if the value of `refcount_t` at the time of the call was nonzero. Internally, the function further guarantees that the increment takes place only when the prior value was in the open interval (0,`UINT_MAX`), thus preventing use after free due to either increment from zero or overflow. This case results in the default return statement at line 20 of Algorithm 4.

[†] http://www.openwall.com/lists/kernel-hardening/2016/11/28/4
[‡] http://elixir.free-electrons.com/linux/v4.10/source/include/net/sock.h}L389
[§] http://elixir:free-electrons:com/linux/v4:10/source/fs/btrfs/raid56:c}L789.

Attempted increment from zero or UINT_MAX results in returns at lines 3 and 6, respectively. Finally, an addition that would overflow the counter instead saturates it by setting its value to UINT_MAX on line 10.

Algorithm 4 refcount_add_not_zero(*refcount*, *summand*) attempts to add a *summand* to the value of *refcount*, and returns true if, and only if, the prior value was non-zero. Note that the function is not locked and only the cmpxchg (line 13) is atomic, i.e., the value of *refcount.value* can change at any point of execution. The loop ensures that the cmpxchg eventually succeeds. Overflow protection is provided by checking if the value is already saturated (line 6 on line 10

```
1:  $val \leftarrow refcount.value$                          ▷ use local copy
2:  while true do
3:      if $val = 0$ then
4:          return false                                   ▷ counter not incremented from zero
5:      end if
6:      if $val =$ UINT_MAX then
7:          return true                                    ▷ counter is saturated, thus not zero
8:      end if
9:      $new \leftarrow val + summand$                      ▷ calculate new value
10:     if $new < val$ then
11:         $new \leftarrow$ UINT_MAX                       ▷ Saturate instead of overflow
12:     end if
13:     $old \leftarrow atomic\_cmpxchg(refcount.value, val, new)$
14:     if $refcount.value$ was unchanged then
15:         $old = val$
16:         break                                          ▷ value was updated by cmpxchg
17:     end if
18:     $val \leftarrow old$                                ▷ update $val$ for next iteration
19: end while
20: return true                                            ▷ value incremented
```

5.5 | Challenges

Despite the additions to the refcount_t API, several reference counting instances nonetheless required careful analysis, and in some cases, modifications to the underlying logic. These challenges were the main reason for not automating the conversion of reference counting schemes using our Coccinelle patterns. The most common challenges are *object pool patterns* and *nonconventional* reference counters.

For example, some implementations of the object pool pattern use negative values of the reference counter to distinguish objects that should be recycled from those that should be freed. Our additions to the refcount_t API support this pattern without resorting to negative reference counter values by using the value of one to indicate that the object should be recycled. These implementations therefore often necessitated nontrivial changes to ensure that neither increment on zero operations nor negative values are expected. Overall, we encountered six particularly challenging recycling schemes. For example, the inoderefcount_t conversion spanned a total of 10 patches.[¶]

In some cases, reference counters were used in nonconventional ways, such as to govern other behavior or track other statistics in addition to the strict reference count itself (eg, the network socket sk_wmem_alloc variable described above). Straightforward conversion to use the refcount_t API can be surprising or outright erroneous for such unconventional uses; it might for instance be expected that such variables can be incremented from zero or potentially reach negative values. In our conversion efforts, we encountered 21 distinct reference counters in this category.

5.6 | Deployment of refcount_t

We developed 233 distinct kernel patches, each converting one distinct variable, spanning all the kernel subsystems. During our work, the refcount_t API, with our additions, was also finalized in the mainline Linux kernel. The next stage of

[¶] http://lkml.org/lkml/2017/2/24/599

our work consisted of submitting all the patches to the respective kernel subsystem maintainers and adapting them based on their feedback. Based on discussions with maintainers, some patches were permanently dropped, either because they would require extensive changes in affected subsystems or would incur unacceptable performance penalty without any realistic risk of actually overflowing the particular counter. As explained in Section 4, some performance-sensitive systems proved challenging to convert due to performance concerns. As a result, a new `CONFIG_REFCOUNT_FULL` kernel configuration option was added to allow switching the `refcount_t` protections off, and thus, use the new API without any performance overhead. This can be utilized by devices that have high-performance requirements but are less concerned about security based on their nature (eg, closed devices such as routers that do not allow installation of untrusted software). These patching efforts, discussions, and patch reviews also uncovered prior reference counting bugs that were fixed in subsequent kernel patches.#

6 | OUT-OF-BOUNDS MEMORY ACCESS

To prevent spatial memory errors in the Linux kernel, we have adapted Intel MPX for in-kernel use. The resulting solution is called MPXK. Although MPX is currently only available in Intel processors, it covers a very large share of existing devices (eg, more than 99% of all servers have Intel processors[43]). Moreover, we hope that, if this technology proves to be successful in preventing spatial memory errors, other CPU vendors would introduce similar technologies. Since the MPX instrumentation (see Section 2.2) utilizes the architecture-independent pointer bounds checker, it should be straightforward to adapt the software components to support similar features from other vendors. The current MPX hardware can be used in both user space and kernel space because it has two distinct sets configuration registers. However, as explained in Section 2.2, current support for MPX is only available for user space applications and requires the assistance of the kernel.‖ Prior to our work, MPX has not been used to protect kernel space.

6.1 | Challenges

The following challenges arise when attempting to use MPX in kernel space.

Memory use: The MPX BD and BTs incur a high memory overhead. As explained in Section 2.2, user space MPX attempts to reduce this overhead by allocating this memory only when needed. However, this requires the kernel to step in at arbitrary points to handle page faults or bound faults caused by BD dereferences or unallocated BTs. This approach cannot be used in the kernel because the kernel cannot handle page faults at arbitrary points *within its own execution*. It is also not feasible to preallocate the BD and BTs, as this would increase base memory usage by over 500%** and require extensive modifications to accommodate certain classes of pointers (eg, pointers originating from user space). An alternative approach would be to substitute the hardware-backed BD and BTs with our own metadata, similar to SoftBound[12] or KASAN.[15] However, this would still incur the same memory and performance overheads as those systems.

Kernel support code: Memory protection extension is supported in user space by GCC library implementations of various support functionality, such as initialization and function wrappers. The user space instrumentation initializes MPX during process startup by allocating the BD in virtual memory and initializing the MPX hardware. However, this existing initialization code cannot be used directly in the Linux kernel because kernel space MPX must be configured during the kernel boot process.

In user space, the compiler also provides instrumented wrapper functions for all memory manipulation functions, such as `memmove`, `strcpy`, and `malloc`. These user space wrappers check incoming pointer bounds and ensure that the correct bounds are associated with the returned pointers. They are also responsible for updating the BD and BTs (eg, `memcopy` must duplicate any BD and BT entries associated with the copied memory). However, these user space wrappers also cannot be used in the kernel because the kernel implements its own standard library.

Binary compatibility (mixed code): User space MPX is binary compatible and can therefore be used in mixed environments with both MPX-enabled code and legacy (noninstrumented) code. A fundamental problem for any binary-compatible bounds checking scheme is that the instrumentation *cannot track pointer manipulation performed by legacy code* and therefore cannot make any assumptions about the pointer bounds after the execution flow returns from

# http://lkml.org/lkml/2017/6/27/567, http://lkml.org/lkml/2017/3/28/383, etc.

‖ Available since the Linux 3.19 kernel release.

**Each potential pointer, ie, any used memory, would require a preallocated BD entry and BT entry, which is the size of four pointers.

the legacy code. Memory protection extension offers a partial solution by storing the pointer's value together with its bounds (using the bndstx instruction) before entering legacy code. When the legacy code returns, MPX uses bndldx to load the bounds again, but since the pointer has been modified, MPX will reset the bounds, essentially making them infinite. This means that pointers changed by legacy code (legitimately or otherwise) can no longer be tracked by MPX.

6.2 | Design of MPXK

In this section, we describe the design of our solution, MPXK, and explain how we solved the aforementioned challenges.

At runtime, MPXK prevents spatial memory errors in the same way as user space MPX, ie, by checking pointer bounds before a pointer is dereferenced. On its base, MPXK determines pointer bounds the same way as MPX, ie, from metadata set by the MPX compile-time instrumentation. However, when the pointer bounds cannot be propagated using the MPX instrumentation (via MPX registers), MPXK does not use the BD and BTs to store and fetch them, but uses its own method, as explained in the *memory use* paragraph as follows. Both user space MPX and MPXK perform narrowing of bounds in exactly the same way, as outlined in Section 2.2.

Memory use: Instead of using BD and BTs for storing and retrieving pointer bounds, MPXK determines them by using *existing kernel metadata*. Specifically, we reuse the kernel memory management metadata created by kmalloc-based allocators that already contain the information about the sizes of allocations. Thus, MPXK does not need to have any additional dynamic storage for the bounds information. We define a new function, mpxk_load_bounds, that queries this existing kernel metadata using a kernel-provided interface to determine the bounds for a pointer allocated by kmalloc, and loads these into the MPX registers. In case a pointer has not been allocated using kmalloc-based allocator, the function returns empty bounds. A side effect of using existing kernel metadata is that retrieved bounds are rounded up to the nearest allocator cache size (ie, may be slightly larger than the requested allocation size). However, this has no security implications because the allocator will not allocate any other objects in the round-up memory area.[44]

Kernel support code: Since we cannot use the existing MPX user space memory management wrappers, we have developed similar kernel-specific wrapper functions that are implemented as normal in-kernel library functions. These MPXK wrappers are significantly less complex (and thus easier to audit for security) than their user space counterparts because the MPXK wrappers do not need to include logic for updating the BD or BTs. In addition to the memory management wrappers, we have also developed new code to initialize the MPX hardware during the kernel boot process. This addresses the second challenge described in Section 6.1.

Binary compatibility: Since in both MPX and MPXK, function arguments rely on the caller to supply bounds, neither scheme can determine bounds for arguments originating from noninstrumented code, and thus, these bounds cannot be checked. As future work, we are investigating how to determine such bounds using the mpxk_load_bounds function. However, since MPXK does not use the bndstx and bndldx instructions but instead attempts to load such bounds using mpxk_load_bounds, it can continue tracking pointers that are modified by legacy code, provided the pointer is supported by mpxk_load_bounds.

Table 1 summarizes the main differences between user space MPX and MPXK.

Kernel instrumentation details

Our MPXK instrumentation is based on the existing MPX support in GCC, but uses our new GCC plugin to adapt this for use in the kernel. Specifically, our plugin instruments the kernel code with the new MPXK bound loading function, MPXK initialization code, and kernel space wrapper functions described previously. We use the GCC plugin system that has been incorporated into Kbuild, the Linux build system since Linux v4.8, to ensure that MPXK is seamlessly integrated with the regular kernel build workflow. To include MPXK instrumentation, a developer simply needs to add predefined

TABLE 1 Summary of the main differences between memory protection extension (MPX) and MPX for kernel (MPXK)

	MPX	MPXK
Hardware initialization	At process start	At kernel boot
Dynamic bounds storage	Uses BD and BTs	Reuses kernel metadata[9]
Memory management function wrappers	Compiler-provided user space wrappers	Kernel's own lightweight wrappers
Pointers modified by legacy code	Cannot be tracked	Can be tracked if supported by mpxk_load_bounds

[9] With the exception of the case, described in Section 7.1 *Indirect pointers* paragraph. Abbreviations: BD, bound directory; BT, bound table.

MPXK flags to any Makefile entries. The plugin itself is implemented in four compiler passes, of which the first three operate on the high-level intermediate representation, GIMPLE, and the last on the lower-level register transfer language, as follows.

1. *mpxk_pass_wrappers* replaces specific memory-altering function calls with their corresponding MPXK wrapper functions, eg, replacing `kmalloc` calls with `mpxk_wrapper_kmalloc` calls.
2. *mpxk_pass_cfun_args* inserts MPXK bound loads for function arguments where bounds are not passed via the four `bndx` registers. This naturally happens when more than four bounds are passed, or due to implementation specifics for any argument beyond the sixth.
3. *mpxk_pass_bnd_store* replaces `bndldx` calls with MPXK bound loads, and removes `bndstx` calls. This covers all high-level (GIMPLE) loads and saves, including return values to legacy function calls.
4. *mpxk_pass_sweeper* is a final low-level pass that removes any remaining `bndldx` and `bndstx` instructions. This pass is required to remove instructions that are inserted during the expansion from GIMPLE to register transfer language.

The source-code for our MPXK-plugin is available at https://github.com/ssg-kernel-memory-safety.

7 | EVALUATION

We evaluate our proposed solutions against the requirements defined in Section 4.

7.1 | Security guarantees

We analyze the security guarantees of both `refcount_t` and MPXK, first, through a principled theoretical analysis, and second, by considering the mitigation of real-world vulnerabilities.

Reference counter overflows

With the exception of `refcount_set` and `refcount_read`, all functions that modify `refcount_t` can be grouped into *increasing* and *decreasing* functions.

All increasing functions maintain the following invariants.

I1: The resulting value will not be smaller than the original value;
I2: A value of zero will not be increased.

The decreasing functions maintain the corresponding invariants.

D1: The resulting value will not be larger than the original value;
D2: A value of `UINT_MAX` will not be decreased.

For example, the increasing function `refcount_add_not_zero` (Algorithm 4) maintains the invariants as follows.

- **input** = 0: Lines 3 and 4 prevent the counter being increased (I2).
- **input** = `UINT_MAX`: Lines 6 and 7 ensure the counter will never overflow (I1).
- **input** ∈ (0,`UINT_MAX`): Lines 10 and 11 ensure that the counter value cannot overflow as a result of addition (I1).

Line 13 ensures that the addition is performed atomically, thus preventing unintended effects if interleaved threads update the counter concurrently. Regardless of how the algorithm exits, the invariant is maintained. The same exhaustive case-by-case enumeration can be used to demonstrate that all other `refcount_t` functions maintain the aforementioned invariants.

An attacker could still attempt to cause a use-after-free error by finding and invoking an *extra decrement* (ie, decrement without a corresponding increment). This is a fundamental issue inherent in all reference counting schemes. However, the errors caused by the extra decrement would almost certainly be detected early in development or testing. In contrast, *missing decrements* are very hard to detect through testing as they may require millions of increments to a single counter before resulting in observable errors. Thanks to the new `refcount_t`, missing decrements can no longer cause reference counter overflows.

In terms of real-world impact, refcount_t would have prevented several past exploits, including CVE-2014-2851, CVE-2016-0728, and CVE-2016-4558. Although it is hard to quantify the current (and future) security impact on the kernel, our observations during the conversion process support the intuition that the strict refcount_t API discourages unsafe implementations. For example, at least two new reference counting bugs[††] were detected and fixed due to their incompatibility with the new API.

As a practical test for the protection provided by the refcount_t type and API, we have tested an attack[‡‡] for the kernel CVE-2016-0728. The exploit abuses a bug inside kernel keyring facility, ie, forgotten decrement of the reference counter when substituting the session keyring with the same keyring. We have made a test on Ubuntu 16.04 with 4.4 mainline kernel with the bug fix commit reverted and converting the corresponding refcounter to use the new refcount_t type. The exploit was successfully stopped (it fails to get the root credentials) and the runtime dmesg log was showing an overflow detected by the refcount_t interface. The corresponding conversion to refcount_t type was one of the first to be merged to the mainline Linux kernel.

Out-of-bounds memory access

The objective of MPXK is to prevent spatial memory errors by performing pointer bounds checking. Specifically, for objects with known bounds, MPXK will ensure that pointers to those objects cannot be dereferenced outside the object's bounds (eg, as would be the case in a classic buffer overflow). A fundamental challenge of bounds checking schemes is thus how to determine the correct bounds for a specific pointer. The MPXK combines compile and runtime mechanisms to load bounds, which means that its ability to load bounds depends on the specific situation.

Static pointers: In the case of local or global pointers to nondynamic memory, all checks and pointer bound handling can be determined and instrumented during compile time. At runtime, such bounds are stored along with the pointers themselves (eg, stack based pointers have stack based bounds) and updated by static instrumentation. The compile-time instrumentation also propagates bounds into and back from function calls. The MPXK can therefore comprehensively perform bounds checking on all static pointers.

Dynamically allocated pointers: Pointers stored in dynamic memory cannot rely on compile-time information and thus require runtime support. The MPXK mpxk_load_bounds function uses available in-kernel memory management metadata to determine the allocated memory area for a specific memory address. Our current implementation can determine bounds for all objects allocated by kmalloc-based allocators and support for other dynamic memory allocators could be similarly added.

Indirect pointers: Indirect pointers, eg, pointers contained by another data structure, are problematic because both their origin and size may be unknown at compile time. If the pointer also points to dynamically allocated memory, mpxk_load_bounds can obtain the bounds as described previously. However, pointers to static memory do not have equivalent runtime metadata and therefore cannot be loaded in this way. Nevertheless, in many cases, these pointers can be covered by the existing FORTIFY_SOURCE directive, which inserts bounds checks based on compile-time-type information.

Pointers in legacy code: Legacy code, ie, code compiled without MPXK support, is by definition not protected by MPXK. Therefore, when pointers originating from legacy code are passed to MPX-enabled code, the bounds are typically not known. However, unlike user space MPX, MPXK can still use mpxk_load_bounds to obtain the bounds if the pointers point to dynamically allocated memory.

Pointer manipulation: If the attacker can corrupt a pointer's value to point to a different object *without dereferencing* the pointer, this can be used to subvert bounds checking schemes. For example, object-centric schemes such as KASAN enforce bounds based on the pointer's value. If this value is changed to point within another object's bounds, the checks will be made (incorrectly) against the latter object's bounds. In theory, pointer-centric schemes such as user space MPX should not be vulnerable to this type of attack, since they do not derive bounds based on the pointer's value. However, for compatibility reasons, if MPX detects that a pointer's value has changed, it *resets* the pointer's bounds (ie, allows it to access the full memory space). Like KASAN, MPXK will use the corrupted value to infer an incorrect set of bounds. The MPXK is therefore no worse than MPX or other object-centric schemes in this regard. However, this type of attack requires the attacker to have a *prior exploit* to corrupt the pointer, which should have been thwarted by MPXK in the first place.

[††] http://lkml.org/lkml/2017/6/27/409, http://lkml.org/lkml/2017/3/28/383
[‡‡] https://www.exploit-db.com/exploits/39277/

TABLE 2 CPU load measurements (in cycles) on different CPUs

Function	Skylake i3-6100U (stddev)	Kaby Lake i7-7500U (stddev)
atomic_inc()	15.1 (0.01)	14.7 (0.07)
refcount_inc()	38.7 (0.03)	49.1 (0.06)

TABLE 3 Netperf refcount usage measurements

Netperf Test type	base (stddev)	refcount (stddev)	change (stddev)
UDP CPU use (%)	0.53 (0.17)	0.75 (0.06)	+42.1% (34%)
TCP CPU use (%)	1.13 (0.03)	1.28 (0)	+13.3% (3%)
TCP throughput (MB/s)	9358 (0)	9305 (0)	-0.6% (0%)
TCP throughput (tps)	14909 (0)	14761 (0)	-1.0% (0%)

Abbreviations: tps, transactions per second; TCP, transmission control protocol; UDP, user datagram protocol.

As a practical demonstration of MPXK's real-world effectiveness, we have tested it against an exploit built around the recent CVE-2017-7184. This vulnerability in the IP packet transformation framework `xfrm` is a classic buffer overflow caused by omission of an input size verification. We first confirmed that we can successfully gain root privileges on a current Ubuntu 16.10 installation running a custom built v4.8 kernel using the default Ubuntu kernel configuration. We then recompiled the kernel applying MPXK on the `xfrm` subsystem, which caused the exploit to fail with a bound violation reported by MPXK.

7.2 | Performance

Reference counter overflows

Although the `refcount_t` functions consist mainly of low-overhead operations (eg, additions and subtractions), they are often used in performance-sensitive contexts. We performed various microbenchmarks of the individual functions during the `refcount_t` development.[§§] The measurements were taken by running the corresponding operation 100 000 times and calculating the average. As shown in Table 2, `refcount_inc` introduces an average overhead of 29 CPU cycles, compared with `atomic_inc`. As shown in Table 2, `refcount_inc` introduces an average overhead of 29 CPU cycles, compared with `atomic_inc`, and the exact number of cycles varies between different tested CPU platforms. The overhead comes from additional overflow and increment-from-zero checks that `refcount_inc` performs similar to `refcount_inc_and_test` explained in detail in Algorithm 1.

However, microbenchmarks cannot be considered in isolation when evaluating the overall performance impact. To gauge the overall performance impact of `refcount_t`, we conducted extensive measurements on the networking subsystem using the Netperf[45] performance measurement suite. We chose this subsystem because (1) it is known to be performance sensitive, (2) it has a standardized performance measurement test suite, and (3) we encountered severe resistance from mainline Linux kernel maintainers on performance grounds when proposing to convert this subsystem to use `refcount_t`. The concerns are well founded due to the extensive use of reference counters (eg, when sharing networking sockets and data) under potentially substantial network loads. The main challenge when evaluating performance impact on the networking subsystem is that there are no standardized workloads, and no agreed criteria as to what constitutes "acceptable" performance overhead. We used the 0-day test service[¶¶] to run the tests on Haswell-EP6 processors and the mainline `v4.11-rc8` kernel. We measured the real-world performance impact of converting all 78 reference counters in the networking subsystem and networking drivers from `atomic_t` to `refcount_t`. Each individual test used a 300-second runtime and was executed three times.

As shown in Table 3, our Netperf measurements include CPU utilization for user datagram protocol (UDP) and transmission control protocol (TCP) streams, and TCP throughput in MB/s and transactions per second. The results indicate that throughput loss is negligible, but the average processing overhead can be substantial, ranging from 13% for UDP to 42% for TCP. First, the results indicate that, for both TCP and UDP, the average processing overhead can be substantial, ranging from 13% for UDP to 42% for TCP. However, rows 3 and 4 in Table 3 show that this overhead does not in practice

[§§] http://lwn.net/Articles/718280/

[¶¶] https://01.org/lkp/documentation/0-day-test-service

TABLE 4 Memory protection extension for kernel (MPXK) and KASAN CPU overhead comparison

	baseline	KASAN		MPXK	
	time in *ns* (stddev)	diff in *ns* (stddev)	% diff	diff in *ns* (stdev)	% diff
No bound load.					
memcpy, 256 B	45 (0.9)	+85 (1.0)	+190%	+11 (1.2)	+30%
memcpy, 65 kB	2340 (4.3)	+2673 (58.1)	+114%	+405 (5.1)	+17%
Bound load needed.					
memcpy, 256 B	45 (0.8)	+87 (0.9)	+195%	+70 (1.5)	+155%
memcpy, 65 kB	2332 (5.8)	+2833 (28.2)	+121%	+475 (15.0)	+20%

affect the overall TCP throughput (both in MB/s and transactions per second). Such processing overhead can in many situations be acceptable; desktop systems typically only use networking sporadically, and when they do, the performance is typically limited by the internet service provider link speed, not CPU bottlenecks. In contrast, this might not be the case in servers or embedded systems (especially routers) where processing resources may be limited and networking a major contributor to system load. However, such systems are typically closed systems, ie, it is not possible to install additional applications or other untrusted software, so therefore their attack surface is already reduced. In these circumstances, the `CONFIG_REFCOUNT_FULL` kernel configuration option can be used to disable the protection offered by `refcount_t` and eliminate all performance overhead. Providing this configuration option is critical to ensure that we can accommodate special cases, such as the aforementioned, while still allowing all other systems to benefit from these new protection mechanisms (which are expected to be enabled by default in the future).

Out-of-bounds memory access

Some CPU overhead is to be expected due to the MPXK instrumentation and bound handling. To measure this, we conducted microbenchmarks of specific functions and end-to-end performance tests with MPXK applied to entire kernel subsystems. All MPXK benchmarks were run on an i3-6100U processor with 8 GB of memory running Ubuntu 16.04 LTS with a v4.8 Linux kernel.

Table 4 shows the results of our microbenchmark on the `memcpy` function, comparing the performance of MPXK and KASAN. The results indicate that, as expected, MPXK does introduce a measurable performance overhead. However, compared with KASAN, the performance overhead is relatively small. The first tests (rows 1 and 2) are conducted for cases in which the bound information is readily available in MPXK from the compiler instrumentation. This is the common case for MPXK and the resulting overhead percentage is much smaller than for KASAN, especially as the size of copied data increases (row 2). The second set of tests (rows 3 and 4) are for the case in which the pointer bounds have to be loaded from the kernel memory management metadata. This is the worst case for MPXK and the largest overhead percentage arises copying small amounts of data (row 3). However, as the size of the copied data increases (row 4), the overall overhead percentage decreases significantly, since only one load of bounds is needed for the whole copied area. Although KASAN also detects temporal memory errors, we controlled for this by only measuring the bounds check events, not the time spent on allocating and deallocating the memory behind a pointer, which is required for implementing protection against temporal memory errors.

In order to demonstrate the performance overhead when MPXK is enabled for a certain kernel subsystem, we have measured the `xfrm` subsystem discussed in Section 7.1. Table 5 shows the result of our end-to-end benchmark in which we measured the impact on an IPSec tunnel where `xfrm` manages the IP package transformations. This was run under the default Ubuntu kernel configuration with our patches applied and MPXK enabled on `xfrm`. We used Netperf for measurements, running five-minute tests for a total of 10 iterations. As shown in Table 5, there is a small reduction in TCP throughput, but the impact on CPU performance is negligible. As shown in Table 5, the impact on CPU performance for both UDP and TCP is negligible, and the resulting TCP throughput is not affected either.

The MPXK also increases runtime memory usage and kernel image size. Although we do not use the costly MPX bound storage, some additional memory is still required to store static global and intermediary bounds. However, our memory use comparisons indicate that this memory overhead is negligible. When deploying MPXK over the `xfrm` subsystem, the memory overhead for in-memory kernel code is 110 KB, which is a 0.7% increase in total size. Similarly, the kernel image size increased by 45 KB or 0.6%. Note that, since MPXK is specifically designed for modular deployment, it can be

TABLE 5 Netperf measurements over an IPSec tunnel with the `xfrm` subsystem protected by memory protection extension for kernel (MPXK)

Netperf test	baseline	MPXK	change (stddev)
UDP CPU use (%)	24.97	24.97	0.00% (0.02)%
TCP CPU use (%)	25.07	25.15	0.31% (0.29)%
TCP throughput (MB/s)	646.69	617.95	-4.44% (4.61)%
TCP throughput (tps)	1586.79	1547.85	-2.45% (1.66)%

Abbreviations: tps, transactions per second; TCP, transmission control protocol; UDP, user datagram protocol.

deployed incrementally to different subsystems, and, if needed, any of the aforementioned overheads can be completely removed for performance-sensitive modules or subsystems.

7.3 | Deployability

Reference counter overflows

Deploying a new data type into a widely used system such as the Linux kernel is a significant real-world challenge. From a usability standpoint, the API should be as simple, focused, and self-documenting as reasonably possible. The `refcount_t` API in principle fulfills these requirements by providing a tightly focused API with only 14 functions. This is a significant improvement over the 100+ functions provided by the `atomic_t` type previously used for reference counting. Of the 233 patches we submitted, 170 are currently accepted, and it is anticipated that the rest will be accepted in the near future. The `refcount_t` type has also been taken into use in independent work by other developers.[##] This is strong confirmation that the usability goals are met in practice. Our contributions also include the Coccinelle patterns for identifying and analyzing reference counting schemes that are potential candidated to be converted to `refcount_t`. These patterns have been contributed by us to the mainline kernel, so that they can be used by other developers and potentially added to the Linux kernel testing infrastructure in the future.

Out-of-bounds memory access

As with user space MPX, our MPXK design considers usability as a primary design objective. It is binary compatible, meaning that it can be enabled for individual translation units. It is also source compatible since it does not require any changes to source code. In a limited number of cases, pointer bounds checking can interfere with valid pointer arithmetic or other atypical pointer use sometimes seen in high-performance implementations. However, such compatibility issues are usually only present in architecture-specific implementations of higher level APIs, and can thus be accommodated in a centralized manner by annotating incompatible functions to prevent their instrumentation. In addition, compatibility issues are usually found during compile time and are thus easily detected during development. The MPXK is fully integrated into Kbuild and provides predefined compilation flags for easy deployment. Using the `MPXK_AUDIT` parameter, MPXK can also be configured to run in permissive mode where violations are only logged, which is useful for development and incremental deployment. This gives the system administrators a choice of when to enable the more restrictive mode, thus facilitating different deployment models.

The MPXK code base is largely self-contained and thus easy to maintain. The only exceptions are the wrapper functions that are used to instrument memory manipulating functions. However, this is not a major concern because the kernel memory management and string APIs are quite stable and changes are infrequent. The MPXK compiler support is all contained within a GCC plugin, and is thus not directly dependent on GCC internals. The MPXK is thus both easy to deploy and easy to maintain.

8 | DISCUSSION

The solutions proposed in this paper and submitted as mainline Linux kernel patches are a step toward improving memory safety in the Linux kernel. One sobering fact when working with production systems and distributed development

[##] http://lkml.org/lkml/2017/6/1/762

communities such as the Linux kernel is that there is always a trade-off between security and deployability. It is certainly possible to propose security solutions outside the mainline kernel, such as the long-lived PaX/Grsecurity[46] patches, but these must then be explicitly applied to production systems. Conversely, solutions that are integrated into the mainline kernel will provide security by default to a large number of diverse devices. In this section, we reflect on our experience of working with kernel developers and subsystem maintainers to integrate our security mechanisms into the mainline kernel. We do not claim to provide a rigorous sociotechnical analysis of this process, but rather we offer a set of recommendations, based on our experience, which we hope could be of value to other researchers and practitioners.

Understand context: It is not sufficient to simply read and follow the Linux kernel contribution guidelines[47] when developing your proposal. It is critical to understand the *context* of the subsystem to which you are attempting to contribute. By context, we mean the recent history and planned developments of the subsystem, the standard ways in which it is verified and tested, and the overall direction set by its maintainers. Presenting your contribution within the correct context removes any initial pushback from other developers and allows the discussion to proceed to the core technical contribution. For example, in our MPXK work, instead of contributing changes directly to the GCC core, we implemented our GCC instrumentation as a standalone GCC plugin (Section 6) using the GCC kernel plugin framework that had been recently added to the mainline Linux kernel. This aligned our contribution with the current direction of the GCC compiler development.

Enable incremental deployability: In the Linux kernel, different subsystems are typically managed by distinct maintainers. Any solutions that require simultaneous changes to all subsystems are thus unlikely to be accepted. This has been cited as a major reason for kernel developers turning down the PaX/Grsecurity solution for reference counters. Instead, solutions that can be incrementally deployed, while still providing benefits, are more likely to receive a favorable reception from at least some developers. The initial deployments can serve as demonstrators to support the case for deployment in other subsystems. For example, the conversion to the new `refcount_t` API can happen independently for each reference counter. Kernel maintainers can thus gradually change their code to use `refcount_t` rather than requiring all kernel code to be changed at once. Similarly, we designed MPXK such that it can be independently enabled for individual compilation units or subsystems.

Provide fine-grained configurability: The Linux kernel runs on a wide range of devices with different environments, threat models, and security requirements. Thus, any security solution (and especially those with performance or usability implications) must support the ability to be configured to several different *grades*. For example, MPXK can be enabled on selected subsystems to provide protection where needed while minimizing performance overhead (see Section 7.3). Similarly, in our reference counter protection work, in addition to having a configuration flag for turning off the protection behind the `refcount_t` interface, there is ongoing work to provide an architecture-specific fast assembly implementation that provides similar security guarantees.[48]

Consider timing: Deploying any new feature that affects more than a single kernel subsystem takes considerable time. This is largely due to the number of different people involved in maintaining various kernel subsystems and the absence of strict organization of the development process. Researchers should plan to allocate sufficient time for this process. In addition, one has to take into account the different stages of the kernel release process in deciding when to send patches to maintainers for review and feedback. For example, it took us a full year to reach the current state of reference counter protection work and have 170 out of 233 patches merged.

Finally, even if a proposed feature is ultimately not accepted, both maintainers and security researchers can learn from the process, which can eventually lead to better security in the mainline kernel.

9 | CONCLUSION AND FUTURE WORK

Securing the mainline Linux kernel is a vast and challenging task. In this paper, we have presented two solutions to proactively memory errors in the kernel, ie, the first prevents temporal memory errors caused by reference counter overflows, while the second provides hardware-assisted checking of pointer bounds to mitigate spatial memory errors. Both solutions treat practical deployability as a primary consideration, with reference counter protection already being widely deployed in the mainline kernel. While our solutions arguably exhibit some limitations, they nonetheless strike a pragmatic balance between deployability and security, thus ensuring they are in a position to benefit the billions of devices using the mainline Linux kernel.

In terms of future work, we will continue our efforts to convert the remaining reference counters to `refcount_t`, but the initial work has already sparked other implementation efforts and a renewed focus on the problem.

There are active efforts to migrate to the mainline kernel solutions that provide high-performance architecture-independent implementations.[48] Several MPXK improvements are also left as future work. Our support for bound loading can be extended with support for other allocators and memory region-based bounds for pointers not dynamically allocated. More invasive instrumentation could be used to improve corner cases where function calls require bound loading. The wrapper implementations could similarly be improved by more invasive instrumentation, or by forgoing wrappers altogether and instead employing direct instrumentation at call sites. This approach is not feasible on vanilla MPX due complications caused by the bound storage, but would be quite reasonable for MPXK.

ACKNOWLEDGEMENTS

We thank the many mainline Linux kernel developers and maintainers who have participated in discussions and patiently provided enlightening feedback and suggestions. This work was supported in part by the Intel Collaborative Research Institute for Secure Computing at Aalto University and the Cloud Security Services (CloSer) project (3881/31/2016), funded by Tekes/Business Finland.

ORCID

Elena Reshetova http://orcid.org/0000-0002-2298-1554

REFERENCES

1. Smalley S, Vance C, Salamon W. Implementing SELinux as a Linux Security Module. 2006. https://www.nsa.gov/resources/everyone/digital-media-center/publications/research-papers/assets/files/implementing-selinux-as-linux-security-module-report.pdf
2. Bauer M. Paranoid penguin: an introduction to Novell AppArmor. *Linux J*. 2006;2006(148):13.
3. Integrity Measurement Architecture (IMA) wiki pages. 2017. http://sourceforge.net/p/linux-ima/wiki/Home/
4. Implementing dm-verity. 2017. http://source.android.com/security/verifiedboot/dm-verity
5. National Vulnerability Database: Statistics for kernel vulnerabilities. 2017. https://nvd.nist.gov/vuln/search/statistics?adv&uscore;search=true&amp;form&uscore;type=advanced&mp;results&uscore;type=statistics&amp;query=kernel
6. Cook K. Status of the Kernel Self Protection Project. 2016. www.outflux.net/slides/2016/lss/kspp.pdf
7. Kernel Self Protection Project wiki. 2017. http://www.kernsec.org/wiki/index.php/Kernel_Self_Protection_Project
8. Raheja S, Munjal G, Shagun. Analysis of Linux kernel vulnerabilities. *Indian J Sci Technol*. 2016;9(48).
9. Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In: Proceedings of the Second Asia-Pacific Workshop on Systems (APSys); 2011; Shanghai, China.
10. Collins GE. A method for overlapping and erasure of lists. *Commun ACM*. 1960;3(12):655-657.
11. McKenney PE. Overview of Linux-kernel reference counting. Technical Report n2167=07-0027. Beaverton, OR: IBM Linux Technology Center; 2007.
12. Nagarakatte S, Zhao J, Martin MMK, Zdancewic S. Softbound: highly compatible and complete spatial memory safety for C. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2009; Dublin, Ireland.
13. Moreira J, Rigo S, Polychronakis M, Kemerlis V. DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel. 2017. https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf
14. Song C, Lee B, Lu K, Harris W, Kim T, Lee W. Enforcing kernel security invariants with data flow integrity. Paper presented: 23rd Annual Network Annual Network and Distributed System Security Symposium (NDSS); San Diego, CA. 2016.
15. The Kernel Address Sanitizer (KASAN). 2017. www.kernel.org/doc/html/v4.10/dev-tools/kasan.html
16. Nikolenko V. Exploiting COF Vulnerabilities in the Linux kernel. 2016. ruxcon.org.au/assets/2016/slides/ruxcon2016-Vitaly.pdf
17. Ramakesavan R, Zimmerman D, Singaravelu P. Intel Memory Protection Extensions (Intel MPX) Enabling Guide. 2015. http://caxapa.ru/thumbs/808589/4878c6471cb5ae28546a594bf25ba5c25c6f.pdf
18. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. In: Proceedings of the Winter 1992 USENIX Conference; 1991; Berkeley, CA.
19. Patil H, Fischer CN. Efficient run-time monitoring using shadow processing. In: Proceedings of the Second International Workshop on Automated Debugging (AADEBUG); 1995; Saint Malo, France.
20. Patil H, Fischer C. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw Pract Exper*. 1997;27(1):87-110.
21. Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the Third International Workshop on Automated Debugging (AADEBUG); 1997; Linköping, Sweden.
22. Yong SH, Horwitz S. Protecting C programs from attacks via invalid pointer dereferences. *ACM SIGSOFT Softw Eng Notes*. 2003;28(5):307-316.

23. Xu W, DuVarney DC, Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *SIGSOFT Softw Eng Notes*. 2004;29(6):117-126.

24. Nethercote N, Fitzhardinge J. Bounds-checking entire programs without recompiling. In: Proceedings of the Second Workshop on Semantics Program Analysis and Computing Environments for Memory Management (SPACE); 2004; Venice, Italy.

25. Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the 28th International Conference on Software Engineering; 2006; Shanghai, China.

26. Necula GC, McPeak S, Weimer W. CCured: type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*. 2002;37(1):128-139.

27. Grossman D, Hicks M, Jim T, Morrisett G. Cyclone: a type-safe dialect of C. *C/C++ Users J*. 2005;23(1):112-139.

28. PaX address space layout randomization (ASLR). 2003. http://pax.grsecurity.net/docs/aslr.txt

29. Branco R. Grsecurity forum — Guest Blog by Rodrigo Branco: PAX_REFCOUNT Documentation. 2015. https://forums.grsecurity.net/viewtopic.php?f=7Zt=4173

30. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2007; San Diego, CA.

31. Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: a fast address sanity checker. In: USENIX Annual Technical Conference; 2012; Boston, MA.

32. Solar Designer. Linux kernel patch to remove stack exec permission. 1997. http://seclists.org/bugtraq/1997/Apr/31

33. Pax non-executable pages design & implementation. 2003. http://pax.grsecurity.net

34. Krahmer S. X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. 2005. https://trailofbits.github.io/ctf/exploits/references/no-nx.pdf

35. Cowan C, Pu C, Maier D, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium; 1998; San Antonio, TX.

36. A stack smashing technique protection tool for Linux. 2011. http://www.angelfire.com/sk/stackshield

37. Xu J, Kalbarczyk Z, Iyer RK. Transparent runtime randomization for security. Paper presented at: 22nd Symposium on Reliable Distributed Systems (SRDS); 2003; Florence, Italy.

38. Ratanaworabhan P, Livshits B, Zorn B. NOZZLE: a defense against heap-spraying code injection attacks. Paper presented at: 18th USENIX Security Symposium; 2009; San Jose, CA.

39. Kwon A, Dhawan U, Smith J, Knight Jr TF, DeHon A. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS); 2013; Berlin, Germany.

40. Kuvaiskii D, Oleksenko O, Arnautov S, et al. SGXBOUNDS: memory safety for shielded execution. In: Proceedings of the Twelfth European Conference on Computer Systems (EuroSys); 2017; Belgrade, Serbia.

41. Devietti J, Blundell C, Martin MMK, Zdancewic S. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGARCH Comput Archit News*. 2008;36(1):103-114.

42. Coccinelle Project. 2017. http://coccinelle.lip6.fr/

43. Pike JP. Server CPU Predictions For 2017. 2017. https://www.forbes.com/sites/moorinsights/2017/01/10/server-cpu-predictions-for-2017/27adb50365a7

44. Akritidis P, Costa M, Castro M, Hand S. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. Paper presented at: 18th USENIX Security Symposium; 2009; San Jose, CA.

45. Netperf Project. 2017. http://hewlettpackard.github.io/netperf

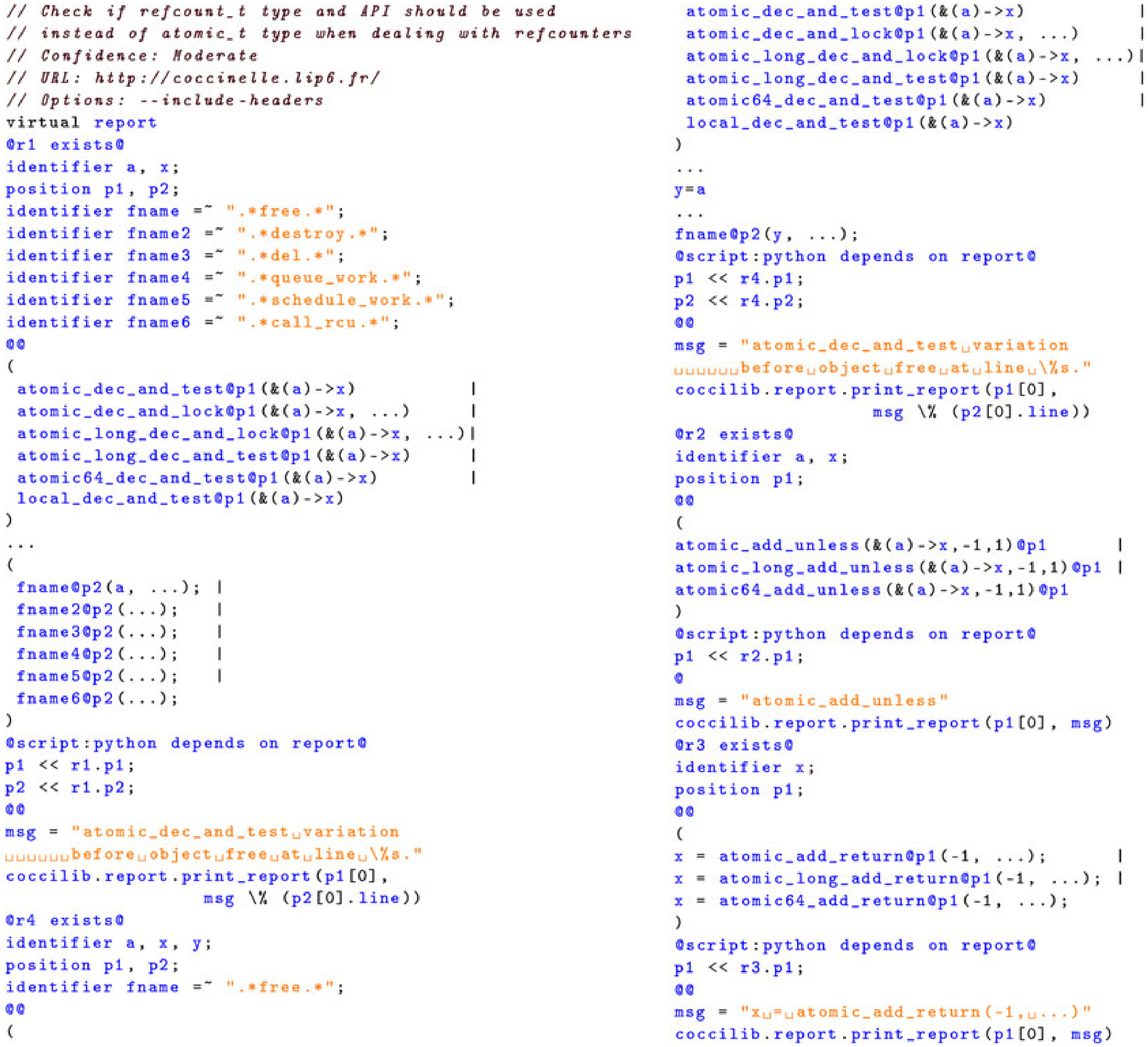
46. Grsecurity Project. 2017. https://grsecurity.net

47. Submitting patches: the essential guide to getting your code into the kernel. 2017. https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html

48. Cook K. codeblog: security things in Linux v4.13. 2017. https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/

How to cite this article: Reshetova E, Liljestrand H, Paverd A, Asokan N. Toward Linux kernel memory safety. *Softw Pract Exper*. 2018;48:2237–2256. https://doi.org/10.1002/spe.2638

APPENDIX



```
// Check if refcount_t type and API should be used
// instead of atomic_t type when dealing with refcounters
// Confidence: Moderate
// URL: http://coccinelle.lip6.fr/
// Options: --include-headers
virtual report
@r1 exists@
identifier a, x;
position p1, p2;
identifier fname =~ ".*free.*";
identifier fname2 =~ ".*destroy.*";
identifier fname3 =~ ".*del.*";
identifier fname4 =~ ".*queue_work.*";
identifier fname5 =~ ".*schedule_work.*";
identifier fname6 =~ ".*call_rcu.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)           |
 atomic_dec_and_lock@p1(&(a)->x, ...)      |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)      |
 atomic64_dec_and_test@p1(&(a)->x)         |
 local_dec_and_test@p1(&(a)->x)
)
...
(
 fname@p2(a, ...); |
 fname2@p2(...);   |
 fname3@p2(...);   |
 fname4@p2(...);   |
 fname5@p2(...);   |
 fname6@p2(...);
)
@script:python depends on report@
p1 << r1.p1;
p2 << r1.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r4 exists@
identifier a, x, y;
position p1, p2;
identifier fname =~ ".*free.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)           |
 atomic_dec_and_lock@p1(&(a)->x, ...)      |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)      |
 atomic64_dec_and_test@p1(&(a)->x)         |
 local_dec_and_test@p1(&(a)->x)
)
...
y=a
...
fname@p2(y, ...);
@script:python depends on report@
p1 << r4.p1;
p2 << r4.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r2 exists@
identifier a, x;
position p1;
@@
(
atomic_add_unless(&(a)->x,-1,1)@p1      |
atomic_long_add_unless(&(a)->x,-1,1)@p1 |
atomic64_add_unless(&(a)->x,-1,1)@p1
)
@script:python depends on report@
p1 << r2.p1;
@
msg = "atomic_add_unless"
coccilib.report.print_report(p1[0], msg)
@r3 exists@
identifier x;
position p1;
@@
(
x = atomic_add_return@p1(-1, ...);      |
x = atomic_long_add_return@p1(-1, ...); |
x = atomic64_add_return@p1(-1, ...);
)
@script:python depends on report@
p1 << r3.p1;
@@
msg = "x␣=␣atomic_add_return(-1,␣...)"
coccilib.report.print_report(p1[0], msg)
```

Listing A1 Coccinelle pattern for finding reference counters in the Linux kernel [Colour figure can be viewed at wileyonlinelibrary.com]

Publication II

Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution, SysTEX '18*, Toronto, ON, Canada, pages 22–47, October 2018.

© 2018 ACM
Reprinted with permission.

Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization

Shohreh Hosseinzadeh*
University of Turku, Finland
shohos@utu.fi

Hans Liljestrand*
Aalto University, Finland
hans.liljestrand@aalto.fi

Ville Leppänen
University of Turku, Finland
ville.leppanen@utu.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

ABSTRACT

Intel Software Guard Extensions (SGX) is a promising hardware-based technology for protecting sensitive computation from potentially compromised system software. However, recent research has shown that SGX is vulnerable to *branch-shadowing* – a side channel attack that leaks the fine-grained (branch granularity) control flow of an *enclave* (SGX protected code), potentially revealing sensitive data to the attacker. The previously-proposed defense mechanism, called *Zigzagger*, attempted to hide the control flow, but has been shown to be ineffective if the attacker can single-step through the enclave using the recent SGX-Step framework.

Taking into account these stronger attacker capabilities, we propose a new defense against branch-shadowing, based on control flow randomization. Our scheme is inspired by Zigzagger, but provides quantifiable security guarantees with respect to a tunable security parameter. Specifically, we eliminate conditional branches and hide the targets of unconditional branches using a combination of compile-time modifications and run-time code randomization. We evaluated the performance of our approach using ten benchmarks from SGX-Nbench. Although we considered the worst-case scenario (whole program instrumentation), our results show that, on average, our approach results in less than 18% performance loss and less than 1.2 times code size increase.

KEYWORDS

Intel SGX; side-channel attack; branch-shadowing attack

ACM Reference Format:
Shohreh Hosseinzadeh, Hans Liljestrand*, Ville Leppänen, and Andrew Paverd. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX, using Control Flow Randomization. In *3rd Workshop on System Software for Trusted Execution (SysTEX '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3268935.3268940

1 INTRODUCTION

Intel Software Guard Extension (SGX)[1] is a recent hardware-based Trusted Execution Environment (TEE) providing isolated execution and guaranteeing the integrity and confidentiality of data within an *enclave*. The enclave is protected from all other software on the platform, including potentially malicious system software (e.g., operating system, hypervisor, and BIOS). Additionally, SGX enables hardware-based measurement and attestation of enclave code.

Although Intel has stated that side-channel attacks are beyond the scope of SGX[2], recent research has demonstrated that SGX is susceptible to several side-channel attacks, which could leak secret information. In particular, Lee et al. [10] demonstrated a *branch-shadowing* side channel attack that allows untrusted software to learn the precise control flow of code running inside an enclave. If this control flow depends on any secret information, this side channel would leak the secret information. This attack abuses the CPU's Branch Prediction Unit (BPU), which is used to improve performance by allowing pipelining of instructions before exact branching decisions are known, i.e., whether or not branches are taken, and the targets of indirect branches. The BPU bases its decisions on recent branch history, which is stored in the CPU's internal Branch Target Buffer (BTB). Two critical factors allow this attack to proceed: 1) BTB entries created by branches inside the enclave are not cleared when the enclave exits; and 2) BTB entries only contain the lower 31 bits of the branch instruction's address, allowing the attacker to create *shadow* branch instructions outside the enclave that map to the same BTB entries as the enclave's branches. The attacker executes the victim enclave, interrupts it immediately after the branch instruction, executes the shadow branch code, and checks whether the branches were correctly predicted, thus revealing whether the BTB entry had been created by the enclave.

Lee et al. [10] also proposed a software-based defense against branch-shadowing, called *Zigzagger*. Using compile-time instrumentation, Zigzagger converts all conditional and unconditional branches into unconditional branches targeting Zigzagger's trampolines, i.e., minimal code sections that hold intermediate jumps — bounces — to the target locations. The Zigzagger trampolines initiate a series of jumps back-and-forth to different branches. The idea is that the attacker cannot interrupt the enclave with sufficient precision to shadow the target branch in this rapid series of jumps. However, SGX-Step [15] invalidates this assumption by

*S. Hosseinzadeh and H. Liljestrand contributed equally to this work, which was done while S. Hosseinzadeh was visiting the Secure Systems Group at Aalto University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SysTEX '18, October 15, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5998-6/18/10…$15.00
https://doi.org/10.1145/3268935.3268940

[1] https://software.intel.com/en-us/sgx

[2] https://software.intel.com/en-us/articles/intel-sgx-and-side-channels

showing how an enclave can be interrupted with single instruction granularity, thus breaking the Zigzagger defense.

The recent Spectre [9] attacks, and their subsequent SGX-specific SGXPectre variant [4] are similar to branch-shadowing in that they exploit the BPU. However, we have confirmed experimentally that neither recent firmware patches, nor the Retpoline compiler-based mitigation affect the ability to perform branch-shadowing attacks.

To overcome this challenge, we present a new defense against branch-shadowing, even if the attacker can single-step through the enclave. Similar to Zigzagger, we use compile-time modifications to convert all branch instructions into unconditional branches targeting our in-enclave trampoline code. At run-time, we then randomize the layout of our trampoline, forcing the attacker to shadow all possible locations. The finite size of the BTB limits the number of guesses the attacker can perform, and thus we can quantify and limit the success probability of a branch-showing attack using the size of the trampoline as a tunable security parameter.

Our contributions are therefore:

- Experimental analysis demonstrating that the recent Spectre mitigation techniques *do not* affect the branch-shadowing attack (Section 3).
- A new approach for defending against branch-shadowing attacks, even in the presence of single-step enclave execution, using control flow randomization (Section 5).
- An initial LLVM implementation of our solution (Section 6)[3] and a quantitative evaluation of its performance and security guarantees (Section 7).

2 BACKGROUND

2.1 Branch Prediction

Intel CPUs use instruction pipelining to load and execute instructions in batches. This allows optimization such as parallelizing and reordering of instructions. The CPU also performs speculative execution, i.e., it uses the BPU to predict which branches will be taken, and executes them before knowing if they are taken.[4] In modern microprocessors, the BPU typically consists of two main subsystems, a BTB and a directional predictor.

The BTB is used to predict the targets of indirect branches.[5] Whenever a branch is taken, a new record is created in the BTB associating the branch instruction's addresses with the target address. Upon encountering subsequent branch instructions, the BPU checks the BTB for the branch instruction address and, if an entry exists, it predicts that the current branch instruction will behave in the same way. The exact details of the BTB lookup algorithms, hashing and size are not public, but the BTB size on Intel Skylake CPUs has been experimentally determined to be 4096 entries [10]. The directional predictor is used to predict whether or not a conditional branch will be taken [5].

Multiple processes executing on the same core share the same BPU, allowing an attacker to misuse the BPU across processes to infer the target and direction of branch instructions [5, 10].

2.2 Intel SGX

Intel SGX is an instruction set extension that provides new instructions to instantiate Trusted Execution Environments (TEEs), called *enclaves*, consisting of code and data. An enclave's data can only be accessed by code running within the enclave, thus protecting it from all other software on the platform, including privileged system software such as the OS or hypervisor. Enclave data is automatically encrypted before it leaves the CPU boundary. However, the OS remains in control of process scheduling and memory mapping, and can therefore control the mapping of (encrypted) enclave memory pages and interrupt enclave execution.

2.3 Branch-shadowing Attacks on SGX

In the *branch-shadowing* attack by Lee et al. [10], the attacker first statically analyzes the unencrypted enclave code and enumerates all branches (i.e., conditional, unconditional, and indirect) together with their target addresses. She then creates shadow code where the branch-instructions and target addresses are aligned such that they will use the same BPU history entries. The attacker then allows the enclave to execute briefly before interrupting it. Finally, she enables the performance counter, in particular the Last Branch Record (LBR), and executes the shadow code, prompting the CPU to predict shadow-branch behavior based on prior enclave execution. The LBR contains information on branch prediction but cannot record in-enclave branches. However, the in-enclave branches can be inferred from the LBR entries for the branches executed after exiting the enclave. Unlike cache-based channels, this does not require timing because the LBR directly reports prediction status.

2.4 Zigzagger and SGX-Step

Zigzagger [10] is presented as a software-based countermeasure to thwart branch-shadowing attack. Zigzagger removes the branches from the enclave functions by obfuscating and replacing a set of branch instructions with a series of indirect jumps. Instead of each conditional branching instruction, an indirect jump and a conditional move (CMOV) is used. Zigzagger assumes that an attacker cannot precisely time the enclave interrupts, i.e., a single probe will cover over 50 instructions. It introduces a trampoline to exercise all unconditional jumps before finally jumping to the final destination. The attacker will typically always detect the same set of taken jumps (i.e., all the unconditional jumps) and cannot distinguish the final jump from the decoy-jumps.

However, Van Bulck et al. [15] presented *SGX-Step*, a framework consisting of a Linux kernel driver and runtime library that manipulates the processor's Advanced Programmable Interrupt Controller (APIC) timer in order to interrupt an enclave after a single instruction i.e., to single-step the enclave's execution. They show that this makes the Zigzagger defense ineffective because the attacker can distinguish meaningful jumps from decoys.

3 SPECTRE MITIGATION TECHNIQUES

The recent Spectre [9] and SGXPectre [4] attacks are similar to branch-shadowing in that they abuse the BPU to exploit speculative execution. Whereas branch-shadowing aims to infer prior branching behavior, these attacks instead manipulate upcoming

[3] Available online at https://github.com/SSGAalto/sgx-branch-shadowing-mitigation

[4] https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[5] https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

branch prediction, e.g., cause speculative execution to touch otherwise inaccessible memory. Although not designed to do so, we suspected the new Spectre mitigation techniques could also affect the branch-shadowing attacks. However, our testing indicates that neither the recent firmware patches from Intel[6], nor the compiler-based Retpoline[7] affect the ability to perform branch-shadowing attacks against SGX.

In particular, we confirmed that Indirect Branch Restricted Speculation (IBRS) — designed to prevent unprivileged code from affecting speculation in privileged execution, e.g., within the enclave — has no effect on branch-shadowing. In our tests we saw no difference between an updated i7-7500U CPU and non-updated machines. We speculate that this is because IBRS is specifically designed to prevent low-privilege code from affecting high-privilege code. Whereas branch-shadowing relies on high-privileged code affecting in subsequent low-privilege code. The Retpoline defense replaces branch instructions with return instructions but our tests indicate that return statements affect the BTB, not only the dedicated Return Stack Buffer (RSB). SGXPectre further demonstrated that Spectre attacks can be performed against Retpoline.

4 THREAT MODEL AND REQUIREMENTS

We assume that the attacker has fine-grained control of enclave execution, i.e., can interrupt the enclave with instruction-level accuracy. The attacker can thus perform a branch-shadowing attack against every branch instruction. Specifically, the attacker can determine whether or not a branch instruction has been executed and taken (i.e., whether a conditional jump fell through or not). If the branching decisions depend on sensitive enclave data, the attacker can infer this data through the branch-shadowing attack.

This is a significantly stronger attacker capability than that assumed by previous work [10] because Van Bulck et al. [15] showed that single-step execution of SGX enclaves is both feasible to implement and sufficient to break existing defenses like Zigzagger [10]. We focus on branch-shadowing attacks and do not consider other side-channels, such as cache or page-fault attacks.

Given these attacker capabilities, we require a defence mechanism that prevents fine-grained branch-shadowing from revealing secret-dependent control flow. Specifically, in the instrumented code, we require that:

R.1 Any branch that can be directly observed through branch shadowing reveals no secret-dependent control flow information.

R.2 For any secret-dependent branches, the attacker's probability of success is bounded based on a security parameter k.

5 PROPOSED APPROACH

Our mitigation scheme uses compile-time obfuscation and run-time randomization to hide the control flow of an enclave application. While our proposed method is inspired by and uses a similar approach to Zigzagger, we assume a stronger attacker model. Specifically, our approach can defend against branch-shadowing even in the presence of an attacker with single-step capabilities.

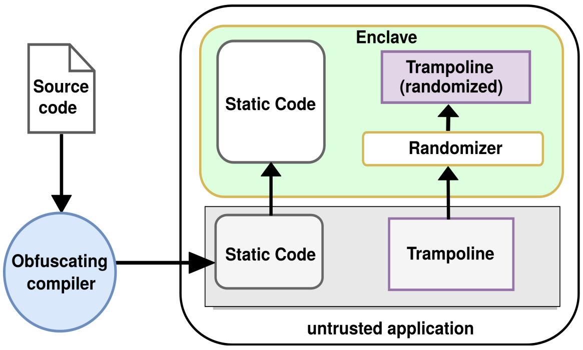


Figure 1: System design

Figure 1 illustrates the high-level view of our approach. The system consists of two main components: an obfuscating compiler and a run-time randomizer. The obfuscating compiler modifies the code by converting all branching instructions to indirect branches. The indirect branch targets are then explicitly set by the instrumentation depending on the converted branch type. We use conditional moves as replacements for conditional branches, allowing us to replicate the functionality of any conditional branch without involving the BPU. The observable control flow transitions, i.e., non trampoline branches, are further organized so that they are always unconditionally executed in the same order. The key insight of our approach is that, unlike Zigzagger, the trampolines are randomized inside the enclave at run-time by the randomizer. This prevents the attacker from reliably tracking their execution. Since only the trampolines are randomized, all other code remains in execute-only memory. Taken together, these two properties fulfill requirements **R.1** and **R.2**, as we show in our security evaluation in Section 7.

Listing 1 and Figure 2 show a single `if`-statement and corresponding Control Flow Graph. The corresponding obfuscated CFG is show in Figure 3. Figure 4 shows the same obfuscated code with the branch instructions converted. The static code is produced at compile time and its layout is assumed to be known to the attacker. The trampoline is similarly produced at compile time but is then randomized at run-time within the enclave. We assume that the attacker can observe and shadow the static code whereas the trampoline is unknown. Specifically, our approach works as follows:

Branch conversion: All branching instructions are converted to indirect unconditional branches. A register (`r15`) is reserved and populated with the original branch targets, which are stored in a jump-table that is updated during randomization. Conditional branches are converted to conditional moves (`cmov`) (e.g., `Block0` in Figure 4).

Jump blocks: Each block is followed by a *jump-block* that jumps to a trampoline indicated by `r15`. Execution flows that do not include a specific block still go through any intermediate jump-blocks to ensure that all indirect jumps outside the trampolines are executed. For instance, when taking the `if`-clause (`Block1`), the else-block (`Block2`) must not be executed but the corresponding jump-block (`B2J`) must be (e.g., the blue line in Figure 4). This ensures that an attacker always sees the same sequence of jumps (i.e., `B0J`, `B1J`, and `B2J`), regardless of actual executed code.

[6] https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[7] https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf

Listing 1: Example code before instrumentation

```
  if (a !=0)
      /* Block1 */
  else
      /* Block2 */
  /* Block3 */
```

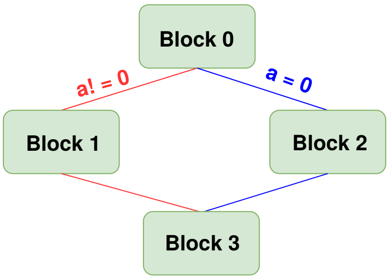


Figure 2: Original control flow graph

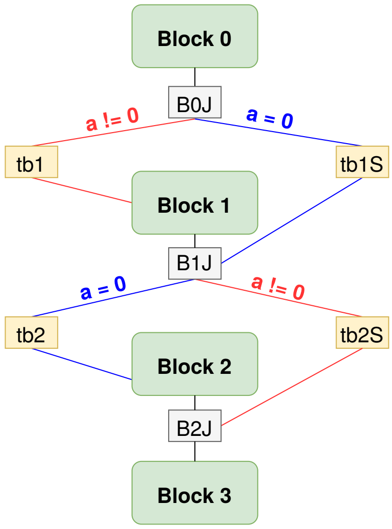


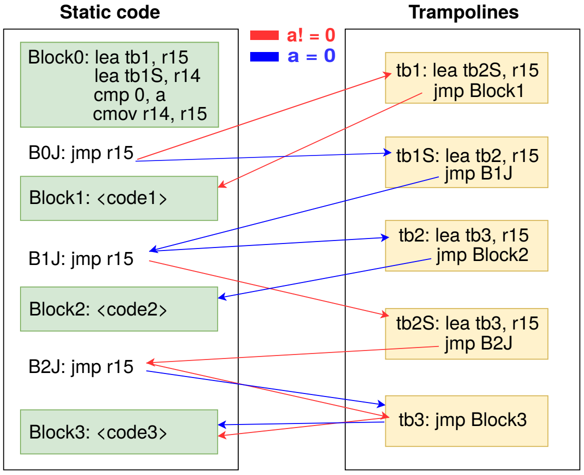
Figure 3: Modified control flow graph

Trampolines: The corresponding trampolines are created, corresponding to either the branching target or the fall-through block (i.e., the next block that will be executed when a conditional branch is not taken). In Figure 3, after execution of the if-block (`Block1`) the control flow is transferred to `tb2S` that will jump to the following jump-block `B2J` without executing the corresponding `Block2` itself.

Skip blocks: When skipping a block — e.g., the `else` block after taking the `if` block — we must nonetheless execute the corresponding jump-block to prevent its omission from leaking information. The jump-block target is prepared in the prior trampoline block by setting `r15`. For instance, after executing the if-block the corresponding trampoline (`tb2S`) not only jumps to the correct jump-block, but also sets the next target, `tb3`, into `r15`. To prevent timing attacks that measure the number of instructions between jump-blocks, the skipping trampolines (e.g., `tb1S` and `tb2S`) are populated with dummy-instructions to ensure that the timing between each jump-block is constant regardless of control flow. Although not shown in our example code, nested blocks are treated similarly to ensure that they execute all intermediary jumps.

Randomization: Trampolines are prepared during compilation, and are randomized at run-time inside the enclave. The randomization is implemented such that shadowing it does not reveal the randomization pattern. Randomizing the trampolines forces the attacker to shadow all possible locations in the enclave and thus, prevents shadowing the trampoline branches and reliably tracking the program's execution.

Re-randomization: Since an attacker could repeatedly call the same enclave functionality to gradually determine the randomization pattern, we can periodically re-randomize the trampolines. For example, the trampolines could be re-randomized on each enclave entry. As future work we envision to: a) provide code-annotation for limiting the obfuscation to only developer-determined sensitive parts, and b) randomize the trampoline code only when detecting multiple enclave entries (i.e., after a given number of potential shadowing attempts).



```
Block0: lea tb1, r15
        lea tb1S, r14
        cmp 0, a
        cmov r14, r15
B0J: jmp r15
Block1: <code1>
B1J: jmp r15
Block2: <code2>
B2J: jmp r15
Block3: <code3>
```

```
tb1: lea tb2S, r15
     jmp Block1
tb1S: lea tb2, r15
      jmp B1J
tb2: lea tb3, r15
     jmp Block2
tb2S: lea tb3, r15
      jmp B2J
tb3: jmp Block3
```

Figure 4: Modified code protected by our approach

6 IMPLEMENTATION DETAILS

We have implemented an open-source prototype of our approach, based on LLVM 6.0 and implemented in the X86 target backend. The instrumentation is applied by systematically traversing all functions and modifying their branching instructions, as explained in Section 5. Since the run-time randomization library cannot be randomized, it must be resistant to branch-shadowing attacks. While implemented, we have not yet integrated the randomizer to our instrumentation. For efficient and fine-grained randomization we do not preform in-place randomization, instead, we move trampoline entries between two trampoline areas. Listing 2 shows an overview of our randomization algorithm. Detailed description is available in our extended technical report [8].

We have also implemented an application for shadowing in-enclave execution in a controlled manner. Our setup is similar to [10] i.e., our application 1) retrieves branch instruction addresses

Listing 2: Randomization algorithm

```
for (entry = 0 : jump_table_entries) {
    location = rand() % trampoline_size;
    if (fits(entry, location))
        mark_reserved(location, entry);
    else
        l = (l+1) % trampoline_size;
    move_entry(entry, location);
}
```

and sets up a corresponding shadow-jump, 2) executes the victim enclave function and returns, 3) enables performance counters and executes the shadow-code, and 4) reads performance counters to infer in-enclave execution. Our setup is such that it could be integrated into the SGXStep-framework. We have replicated the shadowing techniques shown by [10] and performed shadowing on return statements.

7 EVALUATION

7.1 Security Analysis

As specified in Requirements (Section 4), we must prevent an attacker from inferring the secret-dependent control flow by **R.1**) ensuring that observable branches do not leak information, and **R.2**) preventing the attacker from probing other branches with a probability based on the security parameter k.

To hide any data-dependant branches (**R.1**), we replace all conditional branches with unconditional branches. We further setup the control flow so that each block in the static code section is executed in the same order and on each function call. One limitation is that we do not conceal the number of loop executions, because this is typically unknown compile time. In some cases this could be avoided by unrolling loops.

The remaining branching instructions are exclusively in the trampolines, for which the locations are randomized to defend against shadowing (**R.2**). Without knowing the exact trampoline layout, the attacker is forced to guess or exhaustively probe all possible locations. The probability of attack success (P_{attack}) is given by $P_{attack} = \frac{G}{k}$, where G is the number of guesses and k the number of possible trampoline locations.

The upper limit for G is the number of BTB entries, but in practice this is lowered by any intermediate code (e.g., system calls and attack setup) that pollutes the BTB. The security parameter k determines the trampoline randomization space. Because X86 allows unaligned execution, a single 4KB range gives us up to 4091 potential trampoline locations (with a trampoline size starting at 5 bytes). With a randomization area of 8KB and 4096 BTB entries, the success probability of shadowing a single branch has an upper bound of 0.5. The probability of following the full control flow drops exponentially as the number of targeted branches increase.

7.2 Performance Evaluation

We evaluated the overhead of our system in terms of CPU-utilization, memory use, and code size. All software was compiled using the SGX SDK version 2.0 and run on an SGX-enabled Intel Skylake Core i5-6500 CPU clocked at 3.20 GHz, with 7,6 GiB of RAM, running Ubuntu 16.04 with a 64-bit Linux 4.4.0-96-generic kernel.

Table 1: Computational performance (iterations/second) before and after instrumentation, excluding randomization and dummy instructions.

Benchmark	Before (std. dev.)	After (std. dev.)	Performance loss
Numeric sort	828.8 (0.79)	578.8 (0.21)	30%
String sort	86.59 (0.09)	67.72 (0.21)	21%
Bitfield	1.839e8 (1.34e5)	1.370e8 (3.27e5)	25%
Fp emulation	87.70 (0.11)	42.73 (0.02)	51%
Fourier	1.789e5 (1.19e2)	1.500e5 (1.50e2)	16%
Assignment	21.64 (0.03)	7.769 (0.01)	64%
Idea	2667 (1.26)	2665 (1.84)	0.1%
Huffman	2354 (4.07)	860.5 (0.71)	63%
Neural net	35.16 (0.03)	25.57 (0.22)	27%
Lu decomp	973.1 (1.45)	785.0 (1.41)	19%
Geometric mean			17.17%

We used SGX-Nbench[8] which is adapted from Nbench-byte-2.2.3, to measure the CPU and memory overhead of 10 different benchmarks executed within an enclave. All benchmarks were conducted with full instrumentation, but do not include randomization or dummy-instructions. Although the randomization would introduce additional overhead, it need not be constantly repeated. Instead it can be performed once on enclave creation and then later after a specified number of enclave re-entries.

CPU overhead: Table 1 shows the computational performance of various benchmarks in the enclave before and after obfuscation. The decrease in performance (i.e., the number of iterations per second) results from the addition of trampoline jumps and the need to exhaustively execute all jump-blocks. However, since we have obfuscated the entire program, these results represent the worst case scenarios. In real deployments, only the parts of the code that depend on secret data would be obfuscated. The performance penalty depends on how complicated the function is in terms of size and number of branches. The Assignment benchmark, for instance, has functions with many nested conditional branches, all of which require corresponding jump-blocks to be added and executed.

Memory overhead: As expected, our instrumentation does not increase heap or stack usage of the enclave.

Code size: To measure the increase in code size, we compared the size of the enclave object files before and after instrumentation. The size of the SGX-Nbench object files increased from 329.1 kB to 370.1 kB after instrumentation. Similarly to performance overhead, code size overhead will also decrease when instrumenting only the secret-dependent sections of the code.

8 RELATED WORK

There is a growing body of research on side channel attacks targeting Intel SGX and corresponding countermeasures. In addition to the branch-shadowing attacks [5, 10], there are other side channel attacks targeting SGX enclaves [2, 7, 14, 16].

[8] https://github.com/utds3lab/sgx-nbench

Several approaches have been presented to thwart controlled-channel (page-fault) attacks. *SGX-Shield* [11] randomizes the memory layout, similar to Address Space Layout Randomization (ASLR), to prevent control flow hijacking and hide the enclave memory layout. This approach impedes run-time attacks that exploit memory errors or attacks that rely on a known memory layout (e.g., controlled-channel attacks). SGX-Shield uses on-load randomization, allowing repeated branch-shadowing attacks to gradually reveal the randomization pattern. Our approach solves this through run-time re-randomization. We further minimize the additional attack-surface by limiting the randomization to the trampolines.

Shinde et al. [13] propose an approach that masks page-fault patterns by making the program's memory access pattern deterministic. More precisely, they alter the program such that it accesses all its data and code pages in the same sequence, regardless of the input. This makes the enclave application demonstrate the same page-fault pattern for any secret input variables. *T-SGX* [12] leverages Intel Transactional Synchronization Extensions (TSX) to suppress encountered page-faults without invoking the underlying OS. Although T-SGX does not mitigate branch-shadowing attacks [10], it could be combined with our approach to address both branch-shadowing and page-fault attacks.

DR.SGX [1] is presented to defend against cache side-channel attacks. It permutes data locations, and continuously re-randomizes enclave data in order to hamper correlation of memory accesses. This approach prevents leakages resulting from secret-dependant data accesses. Similarly, Chandra et. al [3] inject dummy data instances into the user-supplied data instances in order to add noise to memory access traces. They randomize/shuffle the dummy data with the user data to reduce the chance of extracting sensitive information from side-channels. Both approaches are similar to ours in that they employ randomization, but they are not designed to defend against branch shadowing attacks since they randomize data memory locations rather than control flow.

CCFIR (Compact Control Flow Integrity and Randomization) [17] is a new method proposed to impede control-flow hijacking attacks (e.g., return-into-libc and ROP). CCFIR controls the indirect control transfers and limits the possible jump location to a whitelist in a Springboard. Randomizing the order of the stubs in the Springboard adds an extra layer of protection and frustrates guessing of the function pointers and return addresses. However, CCFIR has not been designed for use in SGX enclaves.

Obfuscation techniques were previously used to thwart leakages via side-channel attacks. Oblivious RAM (ORAM) [6] conceals the program's memory access pattern by shuffling and re-encrypting the accessed data. However, the state should be stored/updated at client-side, which makes it difficult to use for protecting cache since it is challenging to store the internal state of ORAM securely without hardware support, given the small size of cache lines. Moreover, this approach incurs significant performance overhead.

None of the above countermeasures focus on mitigating branch-shadowing attacks, and additionally, Lee et. al [10] have demonstrated that their branch-shadowing attack is capable of breaking the security constructs of SGX-Shield, T-SGX, and ORAM.

9 CONCLUSION AND FUTURE WORK

We propose a software-based mitigation scheme to defend against branch-shadowing attacks, even in the presence of attackers with the ability to single-step through SGX enclaves. Our approach combines compile-time control flow obfuscation with run-time code randomization to prevent the enclave program from leaking secret-dependant control flow. We evaluated our approach using ten benchmarks from SGX-Nbench. Although we considered the worst-case scenario (whole program instrumentation), our results show that, on average, our approach results in less than 18% performance loss and less than 1.2 times code size increase.

As future work, we will integrate the randomizing component and optimize our obfuscating compiler to reduce overhead. In addition, we plan to integrate our approach with other defences, in order to mitigate a broader range of side-channel attacks.

ACKNOWLEDGMENTS

This work was supported in part by the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS) at Aalto University.

REFERENCES

[1] F. Brasser et al. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. (2017). http://arxiv.org/abs/1709.09917

[2] F. Brasser et al. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies*. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[3] S. Chandra et al. 2017. Securing Data Analytics on SGX with Randomization. In *22nd European Symposium on Research in Computer Security*. https://doi.org/10.1007/978-3-319-66402-6_21

[4] G. Chen et al. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. (2018). https://arxiv.org/abs/1802.09085

[5] D. Evtyushkin et al. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3173162.3173204

[6] O. Goldreich and R. Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431 – 473. https://doi.org/10.1145/233551.233553

[7] J. Götzfried et al. 2017. Cache Attacks on Intel SGX. In *10th European Workshop on Systems Security*. https://doi.org/10.1145/3065913.3065915

[8] S. Hosseinzadeh et al. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. (2018). https://arxiv.org/abs/1808.06478

[9] P. Kocher et al. 2018. Spectre Attacks: Exploiting Speculative Execution. (2018). https://spectreattack.com/spectre.pdf

[10] S. Lee et al. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[11] J. Seo et al. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2017.23037

[12] M.-W. Shih et al. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2017.23193

[13] S. Shinde et al. 2015. Preventing Your Faults From Telling Your Secrets: Defenses Against Pigeonhole Attacks. (2015). http://arxiv.org/abs/1506.04832

[14] J. Van Bulck et al. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[15] J. Van Bulck, F. Piessens, and R. Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *2nd Workshop on System Software for Trusted Execution*. https://doi.org/10.1145/3152701.3152706

[16] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.45

[17] C. Zhang et al. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2013.44

Publication III

Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea, Jan-Erik Ekberg, N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, pages 177–195, August 2019.

© 2019 Authors retain copyright
Reprinted with permission.

PAC it up: Towards Pointer Integrity using ARM Pointer Authentication*

Hans Liljestrand
Aalto University, Finland
Huawei Technologies Oy, Finland
hans.liljestrand@aalto.fi

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Kui Wang
Huawei Technologies Oy, Finland
Tampere University of Technology, Finland
wang.kui1@huawei.com

Carlos Chinea Perez
Huawei Technologies Oy, Finland
carlos.chinea.perez@huawei.com

Jan-Erik Ekberg
Huawei Technologies Oy, Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
Aalto University, Finland
asokan@acm.org

Abstract

Run-time attacks against programs written in memory-unsafe programming languages (e.g., C and C++) remain a prominent threat against computer systems. The prevalence of techniques like return-oriented programming (ROP) in attacking real-world systems has prompted major processor manufacturers to design hardware-based countermeasures against specific classes of run-time attacks. An example is the recently added support for *pointer authentication* (PA) in the ARMv8-A processor architecture, commonly used in devices like smartphones. PA is a low-cost technique to authenticate pointers so as to resist memory vulnerabilities. It has been shown to enable practical protection against memory vulnerabilities that corrupt return addresses or function pointers. However, so far, PA has received very little attention as a general purpose protection mechanism to harden software against various classes of memory attacks.

In this paper, we use PA to build novel defenses against various classes of run-time attacks, including the first PA-based mechanism for data pointer integrity. We present PARTS, an instrumentation framework that integrates our PA-based defenses into the LLVM compiler and the GNU/Linux operating system and show, via systematic evaluation, that PARTS provides better protection than current solutions at a reasonable performance overhead.

1 Introduction

Memory corruption vulnerabilities, such as buffer overflows, continue to be a prominent threat against modern software applications written in memory-unsafe programming languages, like C and C++. Theses vulnerabilities can be exploited to overwrite data in program memory. By overwriting control data, such as code pointers and return addresses, attackers can redirect execution to attacker-chosen locations. *Return-oriented programming* (ROP) [35] is a well known technique that allows the attacker to leverage corrupted control-data and pre-existing code sequences to construct powerful (Turing-complete) attacks without the need to inject code into the victim program. By overwriting non-control data, such as variables used for decision making, attackers can also influence program behavior without breaking the program's *control-flow integrity* (CFI) [1]. Such attacks can cause the program to leak sensitive data or escalate attacker privileges. Recent work has shown that non-control-data attacks can also be generalized to achieve Turing-completeness. Such *data-oriented programming* (DOP) attacks [16] are difficult to defend against, and are an appealing attack technique for future run-time exploitation.

Software defenses against run-time attacks can offer strong security guarantees, but their usefulness is limited by high performance overhead, or requiring significant changes to system software architecture. Consequently, deployed solutions (e.g., Microsoft EMET [26]) trade off security for performance. Various hardware-assisted defenses in the research literature [15, 42, 41, 14, 38, 40, 28, 32] can drastically improve the efficiency of attack detection, but the majority of such defenses are unlikely to ever be deployed as they require invasive changes to the underlying processor architecture. However, the prevalence of advanced attack techniques (e.g, ROP) in modern run-time exploitation has prompted major processor vendors to integrate security primitives into their processor designs to thwart specific attacks efficiently [17, 29, 31]. Recent additions to the ARMv8-A architecture [3] include new instructions for *pointer authentication* (PA). PA uses cryptographic message authentication codes (MACs), referred to as *pointer authentication codes* (PACs), to protect the integrity of pointers. However, PA is vulnerable to *pointer reuse* attacks where an authenticated pointer is substituted with another [31]. Practical PA-based defenses must minimize the scope of such substitution.

Goals and Contributions In this work, we further the security analysis of ARMv8-A PA by categorizing pointer

Extended version of this article is available as a technical report [24].

reuse attacks, and show that PA enables practical defenses against several classes of run-time attacks. We propose an enhanced scheme for *pointer signing* that enforces *pointer integrity* for all code and data pointers. We also propose *run-time type safety* which constrains pointer substitution attacks by ensuring the pointer is of the correct type. Pointer signing and run-time type safety are effective against both control-flow and data-oriented attacks. Finally, we design and implement *Pointer Authentication Run-Time Safety* (PARTS), a compiler instrumentation framework that leverages PA to realize our proposed defenses. We evaluate the security and practicality of PARTS to demonstrate its effectiveness against memory corruption attacks. Our main contributions are:

- *Analysis:* A categorization and analysis of pointer reuse and other attacks against ARMv8-A pointer authentication (Section 3).
- *Design:* A scheme for using *pointer integrity* to systematically defend against control-flow and data-oriented attacks, and *run-time type safety*, a scheme for guaranteeing safety for data and code pointers at run-time (Section 5).
- *Implementation:* PARTS, a compiler instrumentation framework that uses PA to realize data pointer, code pointer, and return address signing (Section 6).
- *Evaluation:* Systematic analysis of PARTS showing that it has a reasonable performance overhead ($< 0.5\%$ average overhead for code-pointer and return address signing, 19.5% average overhead for data-pointer signing in nbench-byte (Section 7)) and provides better security guarantees than fully-precise static CFI (9).

We make the source code of PARTS publicly available at `https://github.com/pointer-authentication`.

2 Background

2.1 Run-time attacks

Programs written in memory-unsafe languages are prone to memory errors like buffer-overflows, use-after-free errors and format string vulnerabilities [39]. Traditional approaches for exploiting such errors by corrupting program code have been rendered largely ineffective by the widespread deployment of measures like data execution prevention (DEP). This has given rise to two new attack classes: *control-flow attacks* and *data-oriented attacks* [11].

2.1.1 Control-flow attacks (on ARM)

Control-flow attacks exploit memory errors to hijack program execution by overwriting code pointers (function return addresses or function pointers). Corrupting a code pointer can cause a control-flow transfer to anywhere in executable memory. Corrupting the return address of a function can be used for ROP attacks, which are feasible on several architectures, including ARM [19].

ARM processors, similar to other RISC processor designs, have a dedicated Link Register (LR) that stores the return address. LR is typically set during a function call by the Branch with Link (`bl`) instruction. An attacker cannot directly influence the value of LR, as it is unlikely for a program to contain instructions for directly modifying it. However, nested function calls require the return address of a function to be stored on the stack before the next function call replaces the LR value. While the return address is stored on the stack, an attacker can use a memory error to modify it to subsequently redirect the control flow on function return. On both x86 and ARM, it is possible to perform ROP attacks without the use of return instructions. Such attacks are collectively referred to as *jump-oriented programming* (JOP) [9].

Control-flow integrity (CFI) [1] is a prominent defense technique against control-flow attacks. The goal of CFI is to allow all the control flows present in a program's control-flow graph (CFG), while rejecting other flows. Widely deployed CFI solutions are less precise than state-of-the-art solutions presented in scientific literature.

2.1.2 Data-oriented attacks

In contrast to control-flow attacks, *data-oriented attacks* can influence program behavior without the need to modify code pointers. Instead, they corrupt variables that influence the program's decision making, or leak sensitive information from program memory. Such attacks are called *non-control-data attacks*. Chen et al [11] demonstrated a variety of non-control-data attacks for forging user credentials, changing security critical configuration parameters, bypassing security checks, and escalating privileges. Recent work on DOP [16] showed that non-control-data corruption can also enable expressive attacks without compromising control-flow integrity. DOP may compromise the input of individual program operations and chain together a chosen sequence of operations to achieve the intended functionality.

A data-oriented attack can in principle corrupt arbitrary program objects, but corrupting data pointers is often the preferred attack vector [12]. In Chen et al.'s attack against the GHTTPD web server [11], a stack buffer overflow is used to corrupt a data pointer used in input string validation in order to bypass security checks on the input under the attacker's control. Data pointers are also routinely corrupted in heap exploitation. For instance, the "*House of Spirit*" attack on Glibc[1], involves corrupting a pointer returned by `malloc()` to trick subsequent `malloc()` calls into returning attacker controlled memory chunks. The DOP attacks in [16] also involve the corruption of pointers as a means to control which data is processed by vulnerable code.

[1] Team Shellphish repository of educational heap exploitation techniques: `https://github.com/shellphish/how2heap`

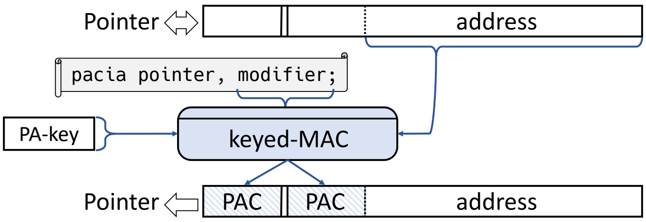


Figure 1: The PAC is created using key-specific PA instructions (pacia) and is a keyed MAC calculated over the pointer address and a modifier.

2.2 ARM Pointer Authentication

ARMv8.3-A includes a new feature called *pointer authentication* (PA). PA is intended for checking the integrity of pointers with minimal size and performance impact. It is available when the processor executes in 64-bit ARM state (AArch64). PA adds instructions for creating and authenticating *pointer authentication codes* (PACs). The PAC is a tweakable message authentication code (MAC) calculated over the pointer value and a 64-bit *modifier* as the tweak (Figure 1). Different combinations of key and modifier pairs allow domain separation among different classes of authenticated pointers. This prevents authenticated pointer values from being arbitrarily interchangeable with one another.

The idea of using of MACs to protect pointers at run-time is not new. *Cryptographic CFI* (CCFI) [25] uses MACs to protect control-flow data such as return addresses, function pointers, and vtable pointers. Unlike ARMv8-A PA, CCFI uses hardware-accelerated AES for speeding up MAC calculation. Run-time software checks are needed to compare the calculated MAC to a reference value. PA, on the other hand, uses either QARMA [5] or a manufacturer-specific MAC, and performs the MAC comparison in hardware.

64-bit ARM processors only use part of the 64-bit address space for virtual addresses (Figure 2). The PAC is stored in the remaining unused bits of the pointer. On a default AArch64 Linux kernel configuration with 39 bit addresses and without address tagging [3, D4.1.4], the PAC size is 24 bits. However, depending on the memory addressing scheme and whether address tagging is used, the size of the PAC is between 3 and 31 bits [31]. Security implications of the PAC size are discussed in Section 9.

PA provides five different keys for PAC generation: two for code pointers, two for data pointers, and one for generic use. The keys are stored in hardware registers configured to be accessible only from a higher privilege level: e.g., the kernel maintains the keys for a user space process, generating keys for each process at process exec. The keys remain constant throughout the process lifetime, whereas the modifier is given in an instruction-specific register operand on each PAC creation and authentication (i.e., MAC verification). Thus it can be used to describe the run-time context in which the pointer is created and used. The modifier value is not necessarily confidential (see Section 4) but ideally such that it 1) precisely describes the context of use in which the pointer is valid, and 2) cannot be influenced by the attacker.

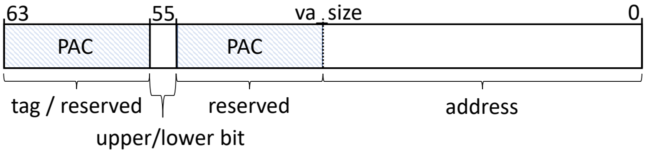


Figure 2: Pointer layout on 64-bit ARM. The PAC is stored in the reserved bits, and its size depends on the used virtual address range. If pointer tagging is disabled, then the PAC can also extend to the tag bits.

PA is used by instrumenting code with PAC creation and authentication instructions. PA instruction mnemonics are generally prefixed either with pac or aut for creation and authentication, respectively, followed by two characters that select one of the data or code keys. For instance, the pacia instruction in Figure 1 will generate an authenticated pointer (pac) based on the instruction (i) A-key (a). Table 5 in Appendix C provides a list of PA instructions referred to in this paper. An authenticated pointer cannot be used directly, as the PAC embedded in the pointer value intentionally interferes with address translation. The corresponding PA authentication instruction (in this case, autia) removes the PAC from the pointer if authentication is successful, i.e., if the current pointer value, key and modifier for autia yields a PAC that matches the PAC embedded in the pointer. If authentication fails, the pointer is invalidated such that a dereference or call using the pointer will cause a memory translation fault. Dedicated PA instructions are encoded in NOP space; older processors without PA support will ignore them.

Return address signing. Qualcomm's return address signing scheme [31] is the first to make use of ARMv8-A PA. It was first introduced in Linaro's GCC toolchain, but has been supported by mainline GCC since version 7.0[2]. It thwarts attacks that manipulate function return addresses through stack corruption (see Section 2.1.1) by ensuring that the return address in LR always contains a PAC when written to or retrieved from memory. Listing 1 shows an example.

The instrumentation adds paciasp (①) at beginning of the function prologue, before the LR value is stored on the stack. paciasp adds a PAC tag using the current Stack Pointer (SP) value as the modifier. Before function return, autiasp (②) authenticates the pointer and either removes the PAC or invalidates the pointer. An alternative is to use the combined autiasp+ret instruction, retaa, but it is not backwards-compatible with older processors.

[2] GCC return address signing and PA support is based on patches provided by ARM, https://github.com/gcc-mirror/gcc/commit/06f29de13f48f7da8a8c616108f4e14a1d19b2c8

```
function:
  paciasp                  ; ① create PAC
  stp FP, LR, [SP, #0]     ; store LR
  ; ...
  ldp FP, LR, [SP, #0]     ; load LR
  autiasp                  ; ② authenticate
  ret                      ; return
```

Listing 1: Return address signing using PA. At funtion entry, `paciasp` is used to create a PAC in LR (①). The value is then authenticated with `autiasp` before return (②).

The PAC cryptographically binds the return address to the current SP value. It is valid only when authenticated using the same SP value as on PAC creation. The goal is to limit the validity of the PAC to the function invocation that created it, thus preventing reuse of authenticated return addresses.

3 Attacks on Pointer Authentication

PA prevents an attacker from injecting or forging pointer values. This effectively prevents any attack that relies on corrupting pointers, resisting even attackers with *arbitrary access to program memory*.

The modifier value used in computing a PAC can depend on both static (e.g., a hard-coded value) and dynamic (e.g., the SP) information. We assume that the program code itself is not confidential and that the attacker can learn how dynamic modifiers are generated and may infer their values.

PA also relies on the security of the underlying cryptographic primitives. In particular, an attacker may attempt to brute-force either the PA keys themselves, or individual PAC values. Sophisticated adversaries may even attempt cryptanalysis attacks based on known PAC values, or side-channels attacks against the hardware circuitry for computing PACs. The security of the QARMA block cipher has already been analyzed [43, 23]. We leave the scrutiny of the cryptographic building blocks outside the scope of this paper. Nevertheless, the limited PAC size means that guessing attacks are a potential concern. We discuss the feasibility of brute-forcing PACs in Section 7.2.4. Assuming proper precautions for the lifetime of PA keys (see Section 2.2), we do not consider guessing attacks the primary attack vector against PA. However, the following concerns for the security of PA-based defenses remain: 1) an attacker controlling the creation of PAC values, or 2) an attacker *reusing* previously authenticated pointers.

Malicious PAC generation. Attackers can potentially control PAC values in three ways, by controlling:

1. *the unauthenticated pointer value before PAC creation*: get an arbitrary authenticated pointer for any context with the same modifier and PA key.
2. *control the PA modifier value*: get an authenticated pointer for a context with the same PA key, but with an attacker-chosen modifier.
3. *both*: get arbitrary authenticated pointers for a context with attacker-chosen modifier, and the same PA key.

To prevent the attacker from generating arbitrary authenticated pointers, the program must not contain PA creation instructions with attacker controlled inputs. Also, a control-flow attack could be mounted by chaining together instruction sequences to prepare the PA operand registers with attacker controlled input and then jump to a PA instruction at another part of the program. This suggests that PA-based defenses *must provide, or be combined with, CFI guarantees* that prevent the use of individual authentication instructions as attacker-controlled gadgets.

Reuse attacks. The attacker can read authenticated pointers (including PAC values), and later reuse them to either:

- *rollback* an authenticated pointer to a previous value, or
- *substitute* an authenticated pointer with another using the same PA modifier.

For instance, in GCC's return address signing scheme (Section 2.2), the return address is bound to the location of the stack frame by using the current SP value as the PA modifier. However, the SP value is not necessarily unique to a specific function invocation. Consequently, an attacker can reuse the authenticated return addresses value from one function when a different vulnerable function executes with a matching SP value. Given that typical programs offer no guarantees on the uniqueness of SP values between different function invocations, this approach exposes a large attack surface for pointer reuse attacks. Therefore, a concern for any PA-based defense is partitioning authenticated pointers into distinct classes based on different <PA key, modifier> pairs.

Attackers can reuse only those pointers they can observe (as opposed all possible values a function pointer can take). Even with full read access to memory (and hence the ability to observe any pointer value that has been generated so far), attackers are still limited to authenticated pointer values the program has already generated.

4 Adversary Model and Requirements

4.1 Pointer Integrity

Kuznetsov et al. [21] introduced the idea of *code pointer integrity*: ensuring precise memory safety for all code pointers in a program. Since control-flow attacks depend on the manipulation of code pointers, guaranteeing code pointer integrity will render *all* control-flow attacks impossible [21].

The notion of *pointer integrity* is generalizable to both code and data pointers. In Section 9.1, we provide a more rigorous definition of pointer integrity. Intuitively, pointer integrity aims to prevent unintentional changes to pointers while they remain in program memory so that the value of a pointer at the time it is "used" (e.g., dereferenced or loaded from memory) is the same as when it was created or stored on memory. In particular, integrity-protected pointers reference the intended target objects. As explained in Section 2.1, all control-flow attacks, all known DOP attacks and many other data-oriented attacks rely on the manipulation of vulnerable pointers. Consequently, ensuring pointer integrity will prevent these attacks.

4.2 Attacker Capabilities

To reason about how effectively PA defends against state-of-the-art attacks we assume attacker capabilities consistent with prior work on run-time attacks (Section 2.1). Our adversary model assumes a powerful attacker with arbitrary memory read and write capabilities restricted only by DEP. The attacker can thus read any program memory and write to non-code segments. We further assume that the attacker has no control of higher privilege levels, i.e., an attacker targeting a user space process cannot access the kernel or higher privilege levels. Specifically, we assume that the attacker cannot infer the PA keys, as they are in registers not directly readable from user space (Section 2.2). We discuss protection of kernel code using PA in Section 10. The attacker's ability to read arbitrary memory precludes the use of randomization-based defenses that cannot withstand information disclosure (e.g., address space layout randomization [36] or software shadow-stacks [1]). PA was specifically designed to remain effective even when the entire memory layout of the victim process is known.

4.3 Goal and Requirements

Our goal is to thwart control-flow and data-oriented attacks by preventing the attacker from forging pointers used by a vulnerable program. We identify the following requirements that our solution should satisfy:

R1 *Pointer Integrity*: Detect/prevent the use of corrupted code and data pointers.

R2 *PA-attack resistance*: Resist attempts to control PAC generation, and pointer reuse attacks.

R3 *Compatibility*: Allow protection of existing programs without interfering with their normal operation.

R4 *Performance*: Minimize run-time and memory overhead and gracefully scale in relation to the number of protected pointers and dereferences/calls.

5 Design

To meet our requirements (Section 4.3) we must solve a number of challenges which we elaborate below.

5.1 Instrument program with PA instructions

To meet requirement R1, the program executable must be instrumented with PA instructions to create and authenticate PACs when needed. For this, we designed and implemented *Pointer Authentication Run-Time Safety* (PARTS), a compiler enhancement that emits PA instructions to sign pointers in memory as required. Specifically, it protects:

- return addresses;
- local, global and static pointers; and
- pointers in C structures.

Figure 3 shows the overall architecture of the PARTS-enhanced compiler. PARTS analyzes the compiler's intermediate representation (IR) to identify any pointers used by the program and then emits PA instructions at points in the program where pointers are (a) created or stored in memory, and (b) loaded from memory or used.

5.2 Create PACs in statically allocated data

Programs may contain pointers which are initialized by the compiler, e.g., defined global variables. However, PAC values for authenticated pointers cannot be calculated before program execution, as PA keys are set only at program launch. Consequently, initialized pointers in the program's data segment pose a challenge, as their values are normally initialized by the linker and loaded into memory separately. PARTS solves this problem by generating a custom initializer function for pointers requiring PACs. At run-time, the PARTS runtime library, PARTSlib, processes the relocated variables and invokes the generated initializer function to ensure that any defined pointers are furnished with a PAC.

5.3 Pointer compartmentalization

As described in Section 3 the attacker may attempt to reuse previously signed pointers. To meet requirement R2 PARTS therefore limits the scope of such reuse attacks by compartmentalizing pointers in three different ways, as shown in Table 1.
Code / Data Pointer Compartmentalization: Recall from Section 2.2, that PA provides separate key sets for data and code pointers making it possible to limit reuse attacks.

Table 1: For code and data pointers PARTS uses a static PA modifier based on the pointer's *ElementType* as defined by LLVM. Return address signing uses a 48-bit `function-id` and the 16 most-significant bits of the SP value.

		key	Modifier type	Modifier construction
①	Data pointer signing	Data A	static	`type-id` = SHA3(ElementType)
②	Code pointer signing	Instr A	static	`type-id` = SHA3(ElementType)
③	Return address signing	Instr B	dynamic + static	SP | `function-id` = compile-time nonce

Run-time type safety: Pointer compartmentalization, while effective, is coarse-grained. To address this, PARTS adds run-time type safety for data and code pointers. Run-time type safety records the pointer's type by encoding it in the PA modifier. Then, it checks that pointer dereferences or indirect calls take place using a pointer with a recorded type that matches the type expected at the use site. PARTS assigns pointers a unique id, `type-id`, based on the pointer's LLVM *ElementType* which depends on the pointed-to data, structure, or function signature. Two pointers are *compatible* (have the same `type-id`) if their ElementType is the same. PARTS uses a deterministic scheme, detailed in Section 6.1 and shown in Table 1, to calculate `type-ids` during compilation. This ensures that separate compilation units generate equivalent `type-ids` for compatible objects, and different `type-ids` for non-compatible ones.

Improved Return Address Signing: While run-time type safety could also be applied for return addresses, it would result in an over-permissive policy for backward edges. As described in Section 3, binding the authenticated return address to the current stack pointer value alone is insufficient because the stack pointer may not be unique to a specific function invocation. Instead, PARTS uses a combination of the current stack pointer value, and a compile-time nonce (`function-id`) ensuring that the authenticated return address cannot be reused across invocations of *different functions*, while the stack pointer values effectively compartmentalizes return addresses to callers with different stack layouts.

5.4 On-load data pointer authentication

Pointers with PACs can be authenticated either as they are loaded from memory, or immediately before they are used. We refer to these as *on-load* and *on-use* authentication, respectively. Data pointers are often dereferenced frequently without intervening function calls, i.e., they will not be cleared after use. This allows the compiler to optimize memory accesses such that, for instance, temporary values might never be written to memory. PARTS accommodates this behavior by only using on-load authentication for data pointers. The combined PA instructions can be used for on-use authentication of code pointers, which are typically loaded to a register, used once, and cleared. On-load authentication always uses the standalone authentication instructions. An attacker could attempt to exploit either the standalone authentication or the separate pointer dereference by diverting control flow to either. However, as mentioned in Section 3, PA solutions must be combined with CFI guarantees, which prevent this type of attacks.

5.5 Handling pointer conversions

A data pointer to an object of a specific type may be converted to a pointer to a different object type. When run-time type safety is applied to authenticated pointers, special care must be taken to not interfere with legitimate pointer conversions to meet requirement $\boxed{\text{R3}}$. For instance, if a struct pointer is cast to a pointer to its first field, it will change the `type-id` and hence the expected PAC.

If the source and destination object types are compatible, no special consideration is needed. If not, PARTS must convert the authenticated pointer to the correct `type-id`. Because data pointer PAC creation and authentication is done at store/load, PARTS handles conversions by; (a) if loading the pointer from memory, validating and stripping the PAC using the `type-id` of the original object, and (b) on store, creating a new PAC using the destination object `type-id`.

A pointer to a function of one type may be converted to a pointer to a function of another type. However, the behavior when calling a function pointer cast to a non-compatible type is undefined [18][6.3.2.3§8]. Hence, PARTS does not need to convert the pointer's PAC to match the destination function's `type-id`. If the converted pointer is converted back, the result is expected to be the same as the original pointer [18][6.3.2.3§8]. PARTS satisfies this as it does not modify the pointer's PAC.

6 Implementation

The PARTS compiler is based on LLVM 6.0 but modifies and adds new passes to the optimizer and the AArch64 backend (Figure 3). The optimization passes (❶) generate necessary metadata for PA modifiers, inserts wrappers for compatibility with legacy code, and prepares initializers for statically allocated pointers. The AArch64 Frame Lowering emits function prologues and epilogues and is modified to include instructions for authenticating the LR value (❷). The

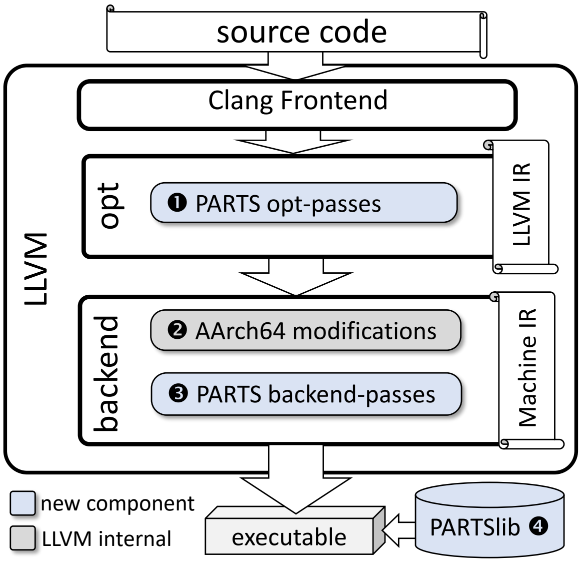


Figure 3: PARTS architecture.

PARTS backend passes (❸) retrieve the PA modifiers and instruments appropriate low-level instructions. The resulting binary is linked with PARTSlib (❹), which at run-time creates PACs for the initialized pointers.

6.1 LLVM Compiler Integration

While the LLVM 6.0 AArch64 backend recognizes PA instructions, they are not used by any pre-existing security feature. Our modifications consist of added optimizer and backend passes, minor modifications to the AArch64 backend, and new PARTS-specific intrinsics. Where applicable, we use optimizer passes that operate on the high-level LLVM *intermediate representation* (IR). Nonetheless, much of the needed functionality is PA-specific and thus implemented in the backend that uses low-level LLVM *machine IR* (MIR), and a register- and instruction set specific to 64-bit ARM.

Determining pointer `type-id`. The compiler backend views the program from a low-level perspective, and the MIR has lost much of the semantics present in C or the high-level IR. Therefore, PARTS must determine `type-ids` during its optimizer passes where this information is still available (Figure 3, ❶). The `type-id` for data consists of a truncated 64-bit SHA-3 hash of the pointer's LLVM `ElementType`. The `ElementType` represents the IR level data type and distinguishes between basic data types, but does not retain `typedef` or other information from the frontend (i.e., clang). Code pointers use the same scheme wherein the `ElementType` consists of the function signature at the same abstraction level. The `type-ids` are passed to the backend either via PARTS-specific compiler intrinsics, or by embedding them as metadata in the existing IR instructions. The AArch64 instruction selection retrieves the information from the IR instructions and transfers it to the emitted MIR (Figure 3, ❷). To facilitate the run-time bootstrap (Section 6.2) PARTS also includes a pass that prepares a custom initializer function that is called at run-time to generate PACs for defined global pointers (Figure 3, ❶).

Return addresses signing. Return address signing is implemented in the AArch64 backend during frame lowering (Figure 3, ❷). Frame lowering emits the function prologues and epilogues, and for non-leaf functions, emits instructions for storing and retrieving the LR value from the stack. PARTS authenticates the value of the LR only if it was retrieved from the stack. The PAC modifier is based on the 16 least-significant bits of the SP value and a 48-bit function-specific `function-id`. The `function-id` is guaranteed to be unique within the current compilation unit or, with link time optimization (LTO), the whole program. To avoid repetition across different compilation units, the `function-id` is generated using a pseudorandom, non-repetitive sequence.

Code pointer signing. PARTS uses the combined PA instructions for branches and converts branch instructions directly to their PA variants (Figure 3, ❸). The PAC for any code pointer is created only once at the time of pointer creation, e.g., when the address of a function is taken. This is instrumented by adding a PAC-creation instruction immediately after the instruction that moves a code pointer to a register. Subsequent load and store operations do not authenticate the signed code pointers, instead they are authenticated only on use.

Data pointer signing. As discussed in Section 5.4, it is not feasible to perform on-use authentication for data pointers. Instead, we authenticate data pointers when they are loaded from memory and create PACs before storing them. In some cases, e.g., using globals, the IR will include explicit load and store operations that can be furnished with the `type-id`. Our modified Instruction Selection then forwards the `type-id` to the emitted MIR (Figure 3, ❷). However, stack-based store and load operations, in particular, are often not present before the backend finalizes the stack-layout and register allocation. Thus, some load and store instructions must be instrumented solely in the backend.

While it would be possible to modify the AArch64 backend (e.g., register allocation), we have instead opted for a less invasive approach. The PARTS backend pass (Figure 3, ❸) finds load and store instructions in the MIR, and uses the attached `type-id` for instrumentation. When the `type-id` is not present, e.g., because the load and store is a register spill, the `type-id` is fetched from surrounding code. For instance, when instrumenting the store due to register spilling

```
MACRO movFunctionId Mod
   movk   Mod, #func_id16, lsl #16
   movk   Mod, #func_id32, lsl #32
   movk   Mod, #func_id48, lsl #48
ENDM

 function:
   mov           Xd, SP              ; ① get SP
   movFunctionId Xd                  ; ② get id
   pacib         LR, Xd              ; ③ PAC
   stp           FP, LR, [SP, #0]    ; store
   ; function body
   ldp           FP, LR, [SP, #0]    ; load LR
   mov           Xd, SP              ; ⑤ get SP
   movFunctionId Xd                  ; ④ get id
   autib         LR, X               ; ⑥ auth
   ret
```

Listing 2: The PARTS return address signing binds the PAC to the SP (①,⑤) and unique function id (②,④). The PA modifier is in register `Xd` during PAC creation (③) and authentication (⑥). The 48-bit `func-id` is split into three 16-bit parts, each moved individually to `Xd` by left-shifting.

a pointer variable, the correct `type-id` can be fetched from the original load.

6.2 Run-time Bootstrap

Programs may contain pointers in statically allocated data, i.e., pointers stored in global variables or static local variables. These are initialized by the compiler or linker, and therefore cannot include PACs. The PARTSlib runtime library instead invokes the compiler generated custom PAC initializer function at process startup. Our Proof-of-Concept implementation invokes the PARTSlib bootstrap using compiler instrumentation that explicitly calls the functionality when entering `main`.

6.3 Instrumentation

PARTS uses only in-line instrumentation and does not require storage of separate run-time metadata. With the exception of the bootstrap process the original code structure is thus largely unchanged. As discussed in Section 2.2, no explicit error handling is added by PARTS; instead, an authentication failure will set specific high-order bits in the pointer, thus triggering a memory translation fault on subsequent dereference or call using the pointer that failed authentication. The high-order bits ensure that the fault is distinguishable as one caused by authentication failure. Our code listings use two macros for setting up PA modifiers for return address signing and `type-id` based PACs, these are shown in Listing 2 and Listing 3.

```
MACRO movTypeId Mod
   mov    Mod, #type_id00
   movk   Mod, #type_id16, lsl #16
   movk   Mod, #type_id32, lsl #32
   movk   Mod, #type_id48, lsl #48
ENDM

mov        cPtr, #instr_addr   ; load cPtr
movTypeId  Xd                  ; ❶ get id
pacia      cPtr, Xd            ; ❷ PAC
; no intermediate cPtr instrumentation
movTypeId  Xd                  ; ❸ get id
blraa      cPtr, Xd            ; ❹ branch
```

Listing 3: The PARTS forward-edge code pointer signing uses the code pointer's `type-id` as the PA modifier (❶,❸). The 64-bit `type-id` is split into four 16-bit parts. The PAC is created only once when initially creating the code pointer (❷). Upon use, i.e., indirect call, the PAC is authenticated using the combined branch and authenticated instruction (❹). PARTS does not instrument intermediate store/load operations.

Return address signing. The return address signing instrumentation is similar to GCC's implementation [31] but includes an added modifier (Listing 2). The function prologue is instrumented such that it prepares the PA modifier by moving SP (①) value into a free register. The SP value is combined with the `function-id` (②) to form the PA modifier, which is then used with the instruction B key (③). The `function-id` is generated at compile-time using LLVM's random number generator, and is guaranteed to be unique withing the LLVM Module (i.e., the whole program, when using link time optimization). The function epilogues (i.e., any part that ends with a return or a tail-call) are similarly instrumented to generate the same PA modifier (④,⑤) and to verify the PAC in the restored LR (⑥).

Code pointer signing. PARTS instruments code pointers only on creation and use (Listing 3). Specifically, when a code pointer is initially created, PARTS will use the instruction A-key to create a PAC (❷) based on the target `type-id` (❶). The instrumentation will at no point remove the PAC from a code pointer. Instead, PARTS uses the combined authenticate and branch instructions — e.g., `blraa` — to perform the branch directly on an authenticated pointer (❹), again using the same PA modifier (❸).

```
ldr     dPtr, [SP, #0]    ; load dPtr
movTypeId Xd, #type_id   ; ① get id
autda dPtr, Xd            ; ② authenticate
; dPtr is directly usable
```

Listing 4: PARTS immediately authenticates data pointers loaded from writeable memory. This is done by first loading the `type-id` (①) and then verifying the PAC (②).

Data pointer signing. All data pointer stores and loads are instrumented such that a PAC is created immediate before store and authenticated immediately after load (Listing 4). When a data-pointer is used the instrumentation first sets up the correct PA modifier, i.e., the `type-id` (①). The pointer is then immediately authenticated using the modifier and data A-key (②); this also strips the PAC from the pointer. As long as the data pointer resides in a register it can thus be used without any performance overhead. PARTS creates PACs for pointers immediately before store in the same manner, save for the `pacda` instruction.

7 Evaluation

We develop our Proof-of-Concept implementation of PARTS on the ARMv8-A Base Platform Fixed Virtual Platform (FVP), based on Fast Models 11.4, which supports version 8.0 to 8.4 of the ARMv8-A architecture [4]. At the time of writing, the only PA-capable hardware is the Apple A12 and S4 SoCs featuring ARMv8.3-A CPUs [2]. However, these proprietary SoCs are, to the best of our knowledge, not available in development versions outside Apple. The FVP provides a software simulation of an ARMv8.3-A processor in AArch64 mode, and is, to the best of our knowledge, the only publicly available environment with ARMv8-A PA support.

7.1 ARMv8.3 Emulation and Software Stack

We use GNU/Linux with a 4.14 kernel, modified to support PA . We modified the bootloader and kernel to activate ARMv8-A PA, and allow key configuration during kernel scheduling at Exception Level 1 (EL1 in Figure 4). Our kernel modifications are based on Mark Rutland's 2018 PA patches[3].

PA keys for each task are stored in a process-specific `mm_context_t` structure (in the process' memory descriptor in the kernel) which contains architecture-specific data related to the process address space. Threads within the same process have a common memory descriptor, and thus share the same PA keys. The scheduler will configure the PA key registers using the keys in the process' memory descriptor whenever a task is scheduled to run. When a new child process is forked, the parent's keys are duplicated to the child's memory descriptor. However, when a new executable file is `exec`'d in the context of an existing process, the kernel initializes a new set of PA keys using `get_random_bytes()`. In other words, each new process receives a new set of PA keys which remain unchanged thereafter.

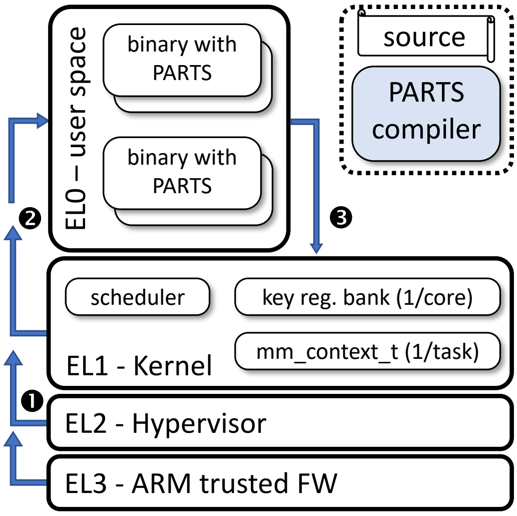


Figure 4: The trapping of PA configuration must be released ❶, in order to allow the kernel to manage the PA keys on process creation and context switches ❷. Faults generated by failed authentications will be trapped by the kernel ❸.

7.2 Security Evaluation

7.2.1 Return address signing

Return address signing in both GCC [31], and PARTS prevents an attacker from introducing forged return addresses to the program stack. Compared to GCC, PARTS augments the PA modifier used for return address signing by combining a function-specific identifier with the SP value ($\boxed{\text{R2}}$). As a result, PARTS return address signing precludes the possibility of reuse of the return address between different functions, irrespective of SP value collisions. It remains susceptible to pointer reuse between distinct invocations of the same function from call sites with same SP value ($\boxed{\text{R1}}$).

7.2.2 Forward-edge code pointer signing

As with PARTS return address signing, forward-edge code pointer signing prevents an attacker from using forged code pointers injected into program memory ($\boxed{\text{R1}}$). This prevents a large class of attacks (e.g., typical ROP/JOP gadgets) that rely on redirecting the control flow to code in the middle of functions, i.e., addresses that never were valid targets of benign control-flow transfers.

[3] `https://lwn.net/Articles/752116/`

PARTS restricts forward-edge code pointer reuse by enforcing *run-time type safety* for signed pointers ($\boxed{\text{R2}}$). Under this scheme, pointers used in a pointer reuse attack must share the same `type-id` (i.e., have a matching type on the LLVM IR level). This prevents large classes of function-reuse attacks. The solution is compatible with common programming patterns involving function pointers ($\boxed{\text{R3}}$), such as callbacks, but allows reuse between code pointers to functions with identical type signatures.

7.2.3 Data pointer signing

PARTS data pointer signing protects all data pointers and prevents an attacker from loading a forged data pointer to program memory ($\boxed{\text{R1}}$). This prevents all non-control data attacks that rely on corrupting data pointers to unintended parts of of memory. This class of attacks includes all currently known DOP attacks [16].

PARTS restricts data pointer reuse by enforcing run-time type safety also for data pointers ($\boxed{\text{R2}}$). Reuse attacks would be more useful to an attacker if they could substitute a vulnerable pointer with one referencing an object of different size or type. Therefore restricting pointer substitution based on the pointer's type restricts the attacker's capability to cause unintended data flows within the program. However, pointer conversions are a challenge for data pointer integrity. As discussed in Section 5.3, PARTS accommodates data pointers that are cast from type A to an incompatible type B by writing the converted pointer using the `type-id` of B. This may expand the effective set of reusable pointers under our threat model; the attacker can record pointers of type A and reuse them at PAC conversion site $A \rightarrow B$, thereby obtaining a pointer of type B to an object of type A. This converted pointer can then be used at de-reference sites that require pointers of type B. If the program also includes a conversion from B to A this makes both types interchangeable.

PARTS data pointer integrity does not guarantee spatial safety of pointer accesses to data objects, nor does it address the temporal safety (e.g., prevent use-after-free conditions). ARMv8-A PA does not provide facilities to directly address these challenges. We discuss orthogonal schemes that can be used in combination with PARTS to provide spatial and temporal safety guarantees in Section 8.

7.2.4 PAC entropy

As explained in Section 3, the PAC size b is a concern for any PA-based scheme. On typical AArch64 Linux systems, b is between 16 and 24. To succeed with probability p, a PAC guessing attack requires $\frac{\log(1-p)}{\log(1-2^{-b})}$ guesses on the assumption that a PAC comparison failure leads to program termination. On our simulator setup where $b = 16$, achieving a 50%-likelihood for a correct guess requires 45425 attempts.

Note that ROP/DOP attacks require an environment where a set of jumps (gadgets) can be set up, each requiring a separate PAC to be broken. Consequently, success probability of a complete attack will decrease exponentially with the number of jumps necessary.

Pre-forked or multithreaded programs will share the same PA key between the parent and all sibling threads/processes. This could allow an attacker to brute force a PAC by targeting a sibling, if PAC failure on a sibling does not result in the termination (and hence PA key reset) of all threads/processes sharing the same PAC key. In this scenario, 2^{b-1} guesses on average are enough to guess a b-bit PAC (32768 guesses for $b = 16$). Multithreaded / pre-forking applications could be hardened against guessing attacks by requiring a full application restart if the number of unexpected terminations of child threads/processes exceeds a pre-defined threshold.

7.3 Performance Evaluation

The FVP processor, peripheral models, and micro-architectural fabric is simplified. Consequently, timing on the FVP model differs from actual hardware. The ARM Fast Models documentation states that *”all instructions execute in one processor master clock cycle“*. We confirm this behavior for PA instructions in the FVP by using microbenchmarks that allow PA instructions to be timed in isolation. As a result, we cannot use the FVP to estimate the expected run-time overhead of PARTS. Instead, we estimate the execution time of PA instructions and develop a PA-analogue that emulates the run-time cost of PA instructions (Section 7.3.1). We then run large-scale benchmarks on real (non-PA) hardware using our PA-analogue (Section 7.3.2).

7.3.1 PA-analogue

From [5, Table 8] we can deduce that on a (1.2GHz) mobile core, the PAC is computable with an approximate overhead of 4 cycles, without accounting for the potential speed benefits of opportunistic pipelining or the inclusion of several parallel PAC computing engines per core. For simplicity, we assume equal cycle counts for all PA instructions. Based on this assumption we construct a *PA-analogue* (Listing 5) as a proxy to measure overhead of PA instrumentation on non-PA CPUs: it consists of four exclusive-or (`eor`) operations to account for the 4 cycles. The final `eor` operates on the modifier and SP to enforce a memory read/write dependency, thus preventing the CPU pipeline from arbitrarily delaying the operations. We have confirmed that our PA-analogue exhibits the expected overhead using our microbenchmarks.

```
eor   Xptr, Xptr, #0x2  ; spend cycles
eor   Xptr, Xptr, #0x3  ; to approximate
eor   Xptr, Xptr, #0x5  ; PA instruction
eor   Xptr, Xptr, Xmod  ; overhead
```

Listing 5: PA-analogue simulating PA instructions

7.3.2 nbench-byte benchmarks

For our performance evaluation we use the Linux nbench-byte 2.2.3 synthetic benchmark[4] designed to measure CPU and memory subsystem performance, providing a reasonable prediction of real-world system performance[5]. We follow work such as [6, 10, 22, 33, 37, 10] and use nbench rather than the SPEC CPU standardized applications benchmarks for our evaluation, as nbench allows us verify the functionality of PARTS instrumentation with manageable simulation times on the FVP. The current version of the SPEC CPU benchmark suite, SPEC CPU2017[6], has replaced many tests in the previous, now retired SPEC CPU2006[7] with significantly larger and more complex workloads (up to ~10X higher dynamic instruction counts). As a result, the SPEC simulation times on the FVP proved to be unmanageable; for example, running individual SPEC benchmarks take hours to *days* to complete on the FVP. This is a challenge for both researchers and industry practitioners who rely on hardware simulation for evaluation [30]. We report our results for a subset of SPEC CPU2017 tests in Appendix B.

The nbench benchmarks include 10 different tests. We adopt the same methodology as Brasser et al. [6] and run each test a constant number of iterations for the following cases: a) uninstrumented baseline b) each PARTS scheme (return address signing, forward-edge code pointer integrity, and data pointer integrity) enabled individually, and c) all schemes enabled simultaneously. Compiler optimizations were disabled for all tests. The tests were performed on a 96boards Kirin 620 HiKey (LeMaker version) with a ARMv8-A Cortex A53 Octa-core CPU (1.2GHz) / 2GB LPDDR3 SDRAM (800MHz) / 8GB eMMC, running the Linux kernel v4.18.0 and BusyBox v1.29.2. Figure 5 shows the results, normalized to the baseline. A more detailed description can be found in Appendix A.

Return address signing incurs a negligible overhead of less than 0.5%. This is expected because the estimated per-function overhead of 12 to 16 cycles is typically small compared to the full execution time of the instrumented function. The same holds for indirect calls (6-8 cycle overhead at the call site), although indirect calls are underrepresented in nbench-byte. However, our microbenchmarks for the code pointer integrity instrumentation indicate that a 6 to 8 cycle overhead per indirect function call is reasonable under the assumed QARMA performance.

Data pointer integrity depends largely on the memory profile of the instrumented program. For instance, the floating point emulation test extensively handles data pointers, resulting in a 39.5% overhead. In contrast, the Fourier and neural network benchmarks contain no data pointers and thus incur no discernible overhead. The geometric mean of the overhead of the combined instrumentation for all tests is 19.5%.

7.4 Compatibility Evaluation

Based on our evaluation, PARTS is compatible with standard C code ($\boxed{\text{R3}}$). Because return address signing only affects the instrumented function, it can be safely applied without interfering with the operation of other parts of programs, or uninstrumented code.

PARTS forward-edge code pointer integrity and data pointer integrity can be safely applied to complete code bases. However, if PARTS is applied only to a partial code base, the instrumented code interfacing with non-instrumented (legacy) libraries requires special consideration. In particular pointers used by both instrumented and uninstrumented code cannot be passed directly between them. We discuss solutions for backwards compatibility with legacy libraries in Section 10.

We encountered no compatibility issues with PARTS during our performance evaluation with nbench (Section 7.3).

8 Related Work

Code-pointer integrity (CPI) [21] protects access to code pointers — and data pointers that may point to code pointers — by storing them in a disjoint area of memory; the *SafeStack*[8]. The SafeStack itself must be protected from unauthorized access. Randomizing the location of the SafeStack is efficient [20], but easily defeated by an attacker who can read arbitrary memory. Stronger protection of the SafeStack using hardware-enforced isolation or software-isolation incurs an average performance overhead of 8.4% or 13.8% in SPEC CPU2006 benchmarks.

Protecting pointers using cryptography. Prior cryptographic defenses against run-time attacks generally assume the attacker cannot read memory. *PointGuard* [12] instruments a program to apply a secret XOR mask to all pointer values. This prevents an attacker from reliably forging pointer values without knowledge of the mask. *Data randomization* [7] extends data masking to cover all data in memory. It uses static points-to analysis and distinct masks to partition memory accesses in separate classes. Neither

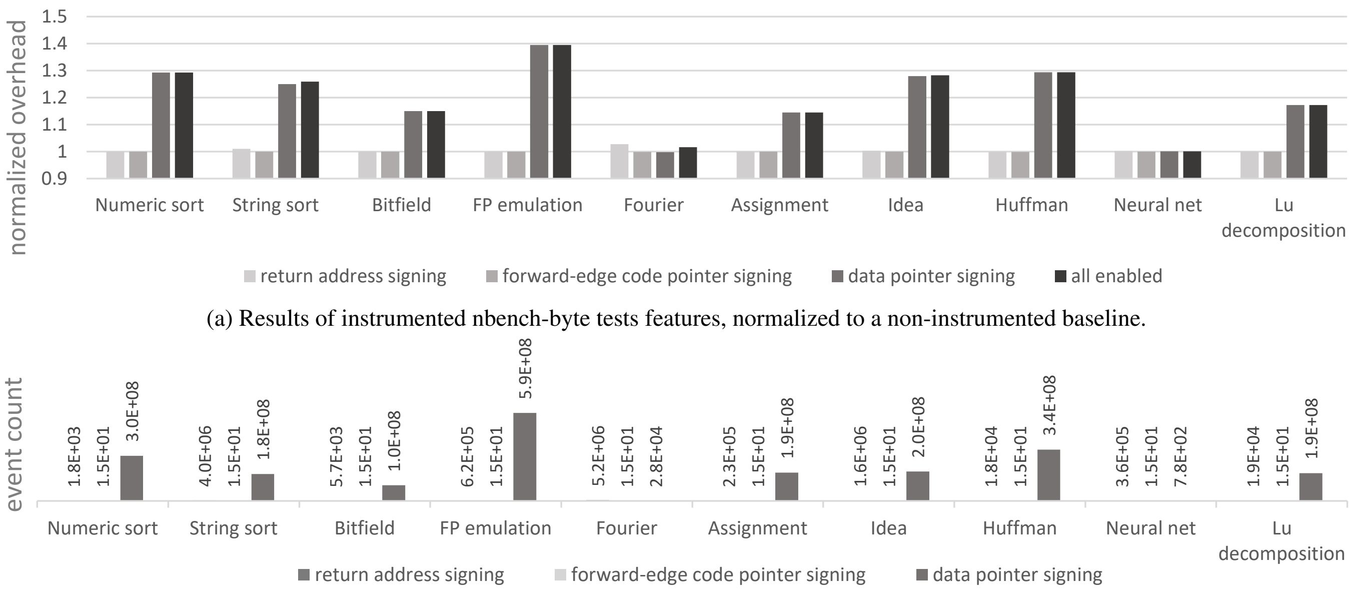
[4] http://www.math.utah.edu/~mayer/linux/bmark.html

[5] http://www.math.utah.edu/~mayer/linux/byte/bdoc.pdf

[6] https://www.spec.org/cpu2017/

[7] https://www.spec.org/cpu2006/

[8] https://clang.llvm.org/docs/SafeStack.html



(a) Results of instrumented nbench-byte tests features, normalized to a non-instrumented baseline.

(b) Run-time count of executed locations instrumentable by PARTS. Because the program's memory profile affects performance the benchmark results clearly correlate with observed memory use (e.g., FP emulation has a large data pointer integrity overhead because it uses many data pointers)

Figure 5: nbench benchmark results

PointGuard nor data randomization remain effective under our threat model.

Similarly to ARMv8-A PA, *Cryptographic CFI* (CCFI) [25] uses MACs to protect control-flow data, such as return addresses, function pointers, and vtable pointers. Like PARTS, CCFI uses a function's type signature to separate function pointers to distinct protection domains, but does not protect function pointers embedded in C structures. Unlike PA, CCFI only benefits from hardware-accelerated AES for speeding up MAC, resulting in a high performance overhead (52% overhead on average in SPEC CPU2006 benchmarks). In contrast, PARTS also benefits from hardware-accelerated checks by using ARMv8-A PA instructions, protects both code and data pointers, including pointers embedded in C structures.

Hardware-assisted mechanisms. Various hardware-assisted defenses are described in research literature [15, 42, 41, 14, 38, 40, 28, 32]. The majority of such defenses have only been realized as soft microprocessor prototypes on FPGAs. Here we describe mechanisms available in commercial off-the-shelf processor architectures.

Only a few commercial processors, such as the SPARC M7[9], support tagged memory, which can be used to realize variety of security models (including pointer integrity). ARM recently announced support for memory tagging in the ARMv8.5-A architecture[10]. It enforces that all accesses to memory must be made via a pointer with the correct tag. Pointer tags use the existing address tagging feature in the ARM ISA that partly overlaps with the bits used to store PA PACs, meaning that enabling both features simultaneously reduces the available PAC size by eight bits.

Hardware-assisted memory tagging is designed primarily as a statistical debug aid against use-after-free and other temporal memory errors. *Hardware-Assisted AddressSanitizer* (HWASAN) [34] is an AArch64-specific compiler-based tool that builds upon *AddressSanitizer* (ASAN) — a memory-error detector popular for vetting memory safety bugs during software testing. ASAN can detect both spatial and temporal memory errors. HWASAN can leverage hardware tagged memory, such as SPARC ADI and the upcoming ARMv8.5-A to reduce the performance overhead associated with managing tagged memory checks in software. ASAN / HWASAN are complementary to PARTS, as they provide spatial and temporal safety for data accesses via pointers.

Intel Memory Protection Extensions (MPX) is a hardware feature for detecting spatial memory errors that debuted in the Intel Skylake microarchitecture. MPX is similar to the software based SoftBound [27] and its hardware-based predecessor [15]. Although Intel MPX is a hardware-assisted approach specifically designed to provide spatial memory safety guarantees, it is not faster than software-based approaches [29]. It can cause up to 4x slowdown in the worst

[9] https://swisdev.oracle.com/_files/What-Is-ADI.html

[10] https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a

case with an average run-time overhead of 50%. It also suffers from other shortcomings, such as the lack of support for multithreading and several common C/C++ idioms. GCC has dropped support for MPX altogether[11].

Control-flow integrity. Carlini et al. [8] define *fully-precise static CFI* as follows: *"An indirect control-flow transfer along some edge is allowed only if there exists a non-malicious trace that follows that edge."* In other words, fully-precise static CFI enforces that execution follows a CFG that contains an edge if and only if that edge is exercised by intended program behavior. Fully-precise static CFI is thus the most restrictive *stateless* policy possible without breaking intended functionality. To date, there exist no implementation of fully-precise CFI; all practical implementations are limited by the precision of CFGs obtained through static *control analysis*.

Carlini et al. further show that all stateless CFI schemes, including fully-precise static CFI are vulnerable to *control-flow bending*; attacks where each control-flow transfer is within a valid CFG, but where the program execution trace conforms to no feasible benign execution trace. For instance, in a stateless policy such as fully-precise static CFI, the best possible policy for return instructions (i.e., backward edges in the CFG) is to allow return instructions within a function F to target any instruction that follows a call to F. In other words, fully-precise static CFI checks if a given control-flow transfer conforms to any of the known control-flow transfers from the current position in the CFG, and does not distinguish between different paths in the CFG that lead to a given control-flow transfer. For this reason CFI is typically augmented with a *shadow call stack* [1, 13] to enforce integrity of return addresses stored on the call stack. We compare PARTS to CFI solutions in Section 9.2.

9 Comparison with other integrity policies

9.1 Fully precise pointer integrity

As discussed in Section 4.1, Pointer Integrity can be loosely defined as a policy ensuring that the value of a pointer at the time of use (dereference or call) corresponds to the value of the pointer when it was created. In this section, we provide a more rigorous definition of Pointer Integrity.

We define *fully-precise pointer integrity* as follows: A pointer dereference is allowed if and only if the pointer is based on its target object. We adopt Kuznetsov et al.'s [21] definition of "*based on*" and say a pointer P *is based* on a target object X if, and only if, P is obtained at run-time by *"(i) allocating X on the heap, (ii) explicitly taking the address of X, if X is allocated statically, such as a local or global variable, or is a control-flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function), (iii) taking the address of a sub-object y of X (e.g., a field in the struct X), or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that are either themselves based on object X or are not pointers."*

Kuznetsov et al's CPI [21] (Section 8) provides fully precise integrity guarantees for *code pointers* by ensuring that accesses to sensitive pointers are safe (sensitive pointers are code pointers and pointers that may later be used to access sensitive pointers). However, CPI requires dedicated, integrity-protected storage for sensitive pointers.

As discussed in Section 7.2, PARTS, and PA solutions in general, achieve an approximation of fully-precise pointer integrity. In particular, PARTS allows the substitution of a pointer P by another pointer P' *based on* object X, if P and P' share the PA modifier. In other words, when PA modifiers are unique to each protected pointer value, PA provides fully-precise pointer integrity. However, ensuring the uniqueness of PA modifiers is not possible in practice due to the following reasons: 1) program semantics may require a set of pointers to be substitutable with each other (e.g., pointers to callback functions) 2) the choice of allowed pointers may depend on run-time properties (e.g., which callback function was registered earlier). In these cases, a unique modifier must be determined at run-time. Fully-precise pointer integrity does *not* imply *memory safety*. In the case of PA, if the modifier is determined at run-time and stored in memory, the PA modifier itself may become a target for an attacker wishing to undermine the integrity policy. To avoid this, modifier values must be derived in a way which leaves the value outside the control of the attacker, e.g., stored in a dedicated hardware register, or read-only program memory.

9.2 Fully-precise static CFI

In contrast to stateless CFI, which allows control-flow transitions present in its CFG regardless of the origin of the code pointer value, PA-based solutions (including PARTS) can preclude forged pointer values from outside the process. The policy that prevents pointer reuse can suffer from limitations similar to those present stateless CFI.

PARTS return address signing provides strong guarantees even when subjected to pointer reuse. In contrast, a stateless CFI policy allows a function to return to any of its call sites. As such, static CFI cannot prevent injection of pointers that are within the expected CFG, i.e., control-flow bending attacks. PARTS additionally requires matching SP values, and that the reused return address originates from a prior function invocation of the same function within the same process for an attack to succeed.

PARTS forward-edge code pointer integrity provides similar guarantees (under reuse attacks) as LLVM's type-based protection (when subjected to any forged pointer). In both

[11] `https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=261304`

cases, attacks are limited to using pointers of the correct dynamic type. PARTS in addition requires that the injected pointer originates from the victim process.

While shadow-stacks protected through randomization can be implemented with minimal performance overhead, our adversary model precludes this approach. Furthermore, software-isolated shadow stack solutions impose impractical performance overheads, and ARM processors do not currently provide direct hardware support for shadow stacks.

10 Conclusion and Future Work

We plan to extend PARTS protection architecture to other protection domains like the OS kernel, or hypervisor. The only significant change for PARTS architecture is to arrange for key configuration for both kernel and EL0 PARTS to be trapped (and managed) on a higher exception level (EL2,3). We are further looking at adding C++ support PARTS. While we do not expect any fundamental problems, some C++ specific features, such as inheritance, cannot be directly handled by our current instrumentation strategy.

Authenticated pointers with PACs cannot be used by legacy code (Section 2.2) while PARTS-instrumented code will trap if pointers without PACs are used. For legacy and PARTS code to interact, we can use wrappers that manipulate function arguments and return values by embedding/stripping PACs. For shared pointers or complex data structures, annotations can disable authentication of selected pointers, allowing programmers to manually adjust pointer conversion to and from legacy code.

Currently, the PARTS compiler assumes shared libraries to be uninstrumented. Instrumented shared libraries must deal with PACs for statically allocated pointers after linking, and thus require changes to the dynamic linker.

Pointer integrity does not imply full memory safety (Section 9.1). Although ARMv8-A PA does not support bounds checking for pointer accesses with authenticated pointers, it has a general-purpose instruction, `pacga`, for producing and validating PACs computed over the contents of two 64-bit registers. This can be used to build authenticated canaries to identify buffer overflow attacks, or to validate the integrity (freshness) of atomic data, such as integer or counter values. In principle, `pacga` instructions can even be chained to validate arbitrary-sized blocks of data.

Finally, effective ways of complementing PA with other emerging memory safety mechanisms like the forthcoming support for memory tagging in ARMv8.5-A is an important line of future work.

Acknowledgments

This work was supported in part by the Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

The authors thank Kostya Serebryany and Rémi Denis-Courmont for interesting discussions and Zaheer Gauhar for implementation assistance.

References

[1] ABADI, M., ET AL. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

[2] APPLE INC. iOS Security — iOS 12. `https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf`, 2018.

[3] ARM LTD. ARMv8 architecture reference manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`, 2017.

[4] ARM LTD. Fast models, version 11.4, fixed virtual platforms (FVP) reference guide. `https://static.docs.arm.com/100966/1104/fast_models_fvp_rg_100966_1104_00_en.pdf`, 2018.

[5] AVANZI, R. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Trans. Symmetric Cryptol. 2017*, 1 (2017), 4–44.

[6] BRASSER, F., ET AL. DR.SGX: Hardening SGX enclaves against cache attacks with data location randomization. `https://arxiv.org/abs/1709.09917`, 2017.

[7] CADAR, C., ET AL. Data randomization. Tech. Rep. MSR-TR-2008-120, Microsoft Research, September 2008.

[8] CARLINI, N., ET AL. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Security '15* (2015), pp. 161–176.

[9] CHECKOWAY, S., ET AL. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.

[10] CHEN, S., ET AL. Detecting privileged side-channel attacks in shielded execution with DéJà Vu. In *Proc. ACM ASIA CCS '17* (2017), pp. 7–18.

[11] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. USENIX Security '05* (2005), pp. 177–191.

[12] COWAN, C., ET AL. PointGuard$^{\text{TM}}$: Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security '03* (2003), pp. 91–104.

[13] DAVI, L., ET AL. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc.NDSS '12* (2012).

[14] DAVI, L., ET AL. HAFIX: Hardware-assisted flow integrity extension. In *Proc. ACM/EDAC/IEEE DAC '15* (2015), pp. 74:1–74:6.

[15] DEVIETTI, J., ET AL. Hardbound: Architectural support for spatial safety of the C programming language. In *Proc. '08* (2008), pp. 103–114.

[16] HU, H., ET AL. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proc. IEEE S&P '16* (2016), pp. 969–986.

[17] INTEL. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2016.

[18] ISO/IEC. ISO/IEC 9899:201x committee draft — December 2, 2010. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`, 2010.

[19] KORNAU, T. *Return Oriented Programming for the ARM Architecture*. PhD thesis, Ruhr-Universität Bochum, 2009.

[20] KUZNETSOV, V., ET AL. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. IEEE S&P '15.

[21] KUZNETSOV, V., ET AL. Code-pointer integrity. In *Proc. USENIX OSDI '14* (2014), pp. 147–163.

[22] LEE, S., ET AL. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. USENIX Security '17* (2017), pp. 557–574.

[23] LI, R., AND JIN, C. Meet-in-the-middle attacks on reduced-round QARMA-64/128. *The Computer Journal 61*, 8 (2018), 1158–1165.

[24] LILJESTRAND, H., ET AL. PAC it up: Towards pointer integrity using ARM pointer authentication. `arXiv:1811.09189 [cs.CR]`, 2019.

[25] MASHTIZADEH, A. J., ET AL. CCFI: Cryptographically enforced control flow integrity. In *Proc. ACM CCS '15* (2015), pp. 941–951.

[26] MICROSOFT. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`, 2016.

[27] NAGARAKATTE, S., ET AL. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. ACM PLDI '09* (2009), pp. 245–258.

[28] NYMAN, T., ET AL. HardScope: Thwarting DOP with hardware-assisted run-time scope enforcement. `arXiv:1705.10295 [cs.CR]`, 2017.

[29] OLEKSENKO, O., ET AL. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. `https://arxiv.org/abs/1702.00719`, 2017.

[30] PANDA, R., ET AL. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon? In *Proc. IEEE HPCA '18* (2018), pp. 271–282.

[31] QUALCOMM TECHNOLOGIES, INC. Pointer authentication on ARMv8.3. `https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf`, 2017.

[32] ROESSLER, N., AND DEHON, A. Protecting the stack with metadata policies and tagged hardware. In *Proc. IEEE S&P '18* (2018), pp. 1072–1089.

[33] SEO, J., ET AL. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proc.NDSS '17* (2017).

[34] SEREBRYANY, K., ET AL. Memory tagging and how it improves C/C++ memory safety. `arXiv:1802.09517 [cs.CR]`, 2018.

[35] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM CCS '07* (2007), pp. 552–561.

[36] SHACHAM, H., ET AL. On the effectiveness of address-space randomization. In *Proc. ACM CCS '04* (2004), pp. 298–307.

[37] SHIH, M.-W., ET AL. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS '17* (2017).

[38] SONG, C., ET AL. HDFI: Hardware-assisted data-flow isolation. In *Proc. IEEE S&P '16* (2016), pp. 1–17.

[39] SZEKERES, L., ET AL. SoK: Eternal war in memory. In *Proc. IEEE S&P '13* (2013), vol. 12, pp. 48–62.

[40] TSAMPAS, S., ET AL. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017* (8 2017), pp. 1–14.

[41] WATSON, R. N. M., ET AL. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. IEEE S&P '15* (2015), pp. 20–37.

[42] WOODRUFF, J., ET AL. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. '14* (2014), pp. 457–468.

[43] ZONG, R., AND DONG, X. Meet-in-the-middle attack on QARMA block cipher. *IACR Cryptology ePrint Archive* (2016).

A nbench experimental setup

The nbench benchmarks employs dynamic workload adjustment to allow the tests to expand or contract depending on the capabilities of the system under test. To achieve this, nbench employs timestamping to ensure that a test run exceeds a pre-determined minimum execution time. If a test run finishes before the minimum execution time has been reached, the test dynamically adjusts its workload, and tries again. For example, the Numeric Sort test will construct an array filled with random numbers, measure the time taken to sort the array. If the time is less than the pre-determined minimum time, the test will build two arrays, and try again. If sorting two arrays takes less time than the pre-determined minimum, the process repeats with more arrays.

Since we want to determine the relative overhead in execution time caused by our instrumentation, we employ the methodology described by Brasser et al. [6] and modify nbench to instead run each test a constant number of iterations. The number of iterations was determined individually for each test based on the iteration counts determined by a unmodified nbench run on the FVP. We then instrument the nbench benchmarks using our PA-analogue (Section 7.3.1) and measure the relative execution time between non-instrumented and instrumented nbench tests on the HiKey development platform using the BusyBox `time` utility.

Each individual benchmark test was run 200 times using the pre-determined number of iterations. Figure 5a, in Section 7.3.2 shows instrumentation overhead for individual tests in relation to the uninstrumented test run. Table 3 shows the numeric overhead ratio for each individual test. Because the nbench benchmarks are designed to measure performance in a manner which is operating system agnostic, they are written in ANSI C and only execute in a single thread. We therefore only consider user time when measuring the overhead of the instrumentation, and exclude context switches and system calls.

The run-time overhead of PARTS is dependent on specific run-time events, such as the number of function invocations in the case of return address signing. Figure 5b in Section 7.3.2 shows the order of magnitude of instrumented run-time events in the nbench tests. We also report the user mode run-time for uninstrumented nbench tests, the number of iterations of each individual test, and number of instrumented run-time events in Table 4.

Table 2: Overhead as ratio and standard deviation (σ) for return address signing and (forward-edge) code pointer signing for 505.mcf_r and 519.lbm_r SPEC benchmarks.

Benchmark	Uninstrumented		ret. addr. sign. + code ptr. integrity	
	ratio	σ	ratio	σ
505.mcf_r	1	0.004	1.005	0.004
519.lbm_r	1	0.000	1.000	0.000

B SPEC CPU2017 experimental setup

Due to unmanageable simulation times in the FVP simulator we have verified the correctness of PARTS instrumentation only on a subset of SPEC CPU2017 benchmarks. Specifically, we chose the 505.mcf_r and 519.lbm_r benchmarks from the SPECrate 2017 integer and floating point suites, because these were the smallest C benchmarks in terms of lines of code. The benchmarks were compiled using SPEC `runcpu`, with a AArch64-specific configuration specifying whole-program-llvm[12], with our PARTS-enabled LLVM, as the compiler. We then extracted the bitcode — created by whole-program-llvm during compilation — and used it to instrument and compile the binaries we used for evaluation: one uninstrumented, one instrumented with PA instructions, and one instrumented with our PA-analogue. We enabled both return address and forward-edge code pointer signing for the instrumented binaries.

We run the PARTS-instrumented binaries on the FVP simulator to confirm correct functionality. The simulation time for the tested benchmarks was between 12 and 48 hours. Performance benchmarks, for baseline and PA-enabled binaries, were run on the HiKey devices, using the same setup as our nbench evaluation. The results are shown in Table 2, and are based on five runs of each benchmark. In 505.mcf_r we observed overheads consistent with our results from nbench. We observed no discernible overhead in 519.lbm_r. We attributed this to the following properties of 519.lbm_r: (a) it does not exhibit forward-edge code pointers, and (b) it has few non-leaf function calls in relation to the arithmetic computation performed part of the benchmark.

[12] `https://github.com/travitch/whole-program-llvm`

Table 3: Overhead as ratio and standard deviation (σ) for nbench tests reported separately for uninstrumented, return address signing, (forward-edge) code pointer signing, data pointer signing and all instrumentation enabled.

Test	Uninstrumented		PARTS							
			ret. addr. sign		code ptr. signing		data ptr. signing		all enabled	
	ratio	σ	ratio	σ	ratio	σ	ratio	σ	ratio	σ
Numeric sort	1	0.002	1	0.003	1	0.003	1.293	0.003	1.293	0.003
String sort	1	0.002	1.01	0.002	1	0.002	1.251	0.002	1.259	0.002
Bitfield	1	0.002	1	0.002	1	0.002	1.15	0.002	1.15	0.001
FP emulation	1	0.001	1	0.001	1	0.001	1.395	0.001	1.396	0.001
Fourier	1	0.002	1.027	0.004	0.999	0.003	0.998	0.002	1.016	0.003
Assignment	1	0.001	1	0.002	1	0.002	1.145	0.002	1.145	0.002
Idea	1	0.001	1.004	0.002	1	0.002	1.279	0.002	1.283	0.002
Huffman	1	0.001	0.999	0.001	0.999	0.001	1.294	0.001	1.295	0.002
Neural net	1	0.001	1.002	0.002	1	0.002	1.001	0.002	1.001	0.003
Lu decomposition	1	0.001	1	0.002	1	0.002	1.173	0.002	1.173	0.002
Geometric average	1	-	1.004	-	1.000	-	1.191	-	1.195	-

Table 4: User mode run-time (utime) and standard deviation (σ) in seconds for uninstrumented nbench tests, the pre-determined number of iterations for each individual test, and the number of run-time events that are affected by instrumentation. Non-leaf calls correspond to function invocations protected by return address signing. Leaf calls correspond to function invocations which do no store the value of LR in memory, and thus can be left uninstrumented. Instruction pointers created and indirect calls are instrumented by (forward-edge) code pointer signing, and data pointer loads / stores correspond to events where data pointer instrumentation is active.

Test	Baseline			Instrumented events				
	utime	σ	iterations	non-leaf calls	leaf calls	instr. ptr. created	indirect calls	data ptr. ldr/str
Numeric sort	3.573	0.007	350	1802	7117598	10	5	302212833
String sort	2.971	0.005	125	3977237	1022510	10	5	180105579
Bitfield	2.687	0.004	101647890	5669	4308	10	5	104670943
FP emulation	5.862	0.004	35	616536	37906118	10	5	589518589
Fourier	2.693	0.005	25870	5240188	161	10	5	27504
Assignment	4.414	0.005	10	225602	113353	10	5	190662093
Idea	2.808	0.004	1500	1640184	54420196	10	5	196844406
Huffman	4.212	0.005	1000	17659	46983276	10	5	343176061
Neural net	5.477	0.007	10	359423	441412	10	5	782
Lu decomposition	3.596	0.005	230	18970	441412	10	5	186704928

C ARMv8-A PA Instructions

Table 5: List of PA instructions referred to in the main paper [3]. *PA Key* indicates the PA key the instruction uses. *Addr.* indicates the source of the address to be signed / authenticated (*Xd* indicates that the address is specified using a general purpose register). *Mod.* indicates the modifier used by the instruction (*Xm* indicates that the modifier is specified by a general purpose register.) The *backwards-compatible* column indicates if the instruction encoding resides in the NOP space for pre-existing ARMv8-A processors.

Instruction	Mnemonic	PA Key					Addr.	Mod.	Backwards-compatible
		Instr.		Data		Gen-eric			
		A	B	A	B				
BASIC POINTER AUTHENTICATION INSTRUCTIONS									
Add PAC to instr. addr.	`paciasp`	✓					LR	SP	✓
	`pacia`	✓					*Xd*	*Xm*	✓
	`pacibsp`		✓				LR	SP	✓
	`pacib`		✓				*Xd*	*Xm*	✓
Add PAC to data addr.	`pacda`			✓			*Xd*	*Xm,*	✓
	`pacdb`				✓		*Xd*	*Xm*	✓
Calculate generic MAC	`pacga`					✓			✓
Authenticate instr. addr.	`autiasp`	✓					LR	SP	✓
	`autia`	✓					*Xd*	*Xm*	✓
	`autibsp`		✓				LR	SP	✓
	`autib`		✓				*Xd*	*Xm*	✓
Authenticate data addr.	`autda`			✓			*Xd*	*Xm,*	✓
	`autdb`				✓		*Xd*	*Xm*	✓
COMBINED POINTER AUTHENTICATION INSTRUCTIONS									
Authenticate instr. addr. and return	`retaa`	✓					LR	SP	✗
	`retab`		✓				LR	SP	✗
Authenticate instr. addr. and branch	`braa`	✓					*Xd*	*Xm*	✗
	`brab`		✓				*Xd*	*Xm*	✗
Authenticate instr. addr. and branch with link	`blraa`	✓					*Xd*	*Xm*	✗
	`blrab`		✓				*Xd*	*Xm*	✗
Authenticate instr. addr. and exception return	`eretaa`	✓					ELR	SP	✗
	`eretab`		✓				ELR	SP	✗
Authenticate data. addr. and load register	`ldraa`				✓		*Xd*	*zero*	✗
	`ldrab`					✓	*Xd*	*zero*	✗

Publication IV

Hans Liljestrand, Zaheer Gauhar, Thomas Nuyman, Jan-Erik Ekberg, N. Asokan. Protecting the stack with PACed canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution, SysTEX '19*, Huntsville, ON, Canada, 6 pages, October 2019.

© 2019 ACM
Reprinted with permission.

Protecting the stack with PACed canaries

Hans Liljestrand
Aalto University, Finland
Huawei Technologies Oy, Finland
hans@liljestrand.dev

Zaheer Gauhar
Aalto University, Finland
zaheer.gauhar@aalto.fi

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Jan-Erik Ekberg
Huawei Technologies Oy, Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
University of Waterloo, Canada
asokan@acm.org

Abstract

Stack canaries remain a widely deployed defense against memory corruption attacks. Despite their practical usefulness, canaries are vulnerable to memory disclosure and brute-forcing attacks. We propose PCan, a new approach based on ARMv8.3-A pointer authentication (PA), that uses dynamically-generated canaries to mitigate these weaknesses and show that it provides more fine-grained protection with minimal performance overhead.

CCS Concepts • **Security and privacy** → *Embedded systems security.*

ACM Reference Format:
Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. 2019. Protecting the stack with PACed canaries. In *4th Workshop on System Software for Trusted Execution (SysTEX '19), October 27, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3342559.3365336

1 Introduction

Run-time attacks that exploit memory errors to corrupt program memory are a prevalent threat. Overflows of buffers allocated on the stack are one of the oldest known attack vectors [12, 17]. Such exploits corrupt local variables or function return addresses. Modern attacks techniques—such as return-oriented programming (ROP) [16] and data-oriented programming (DOP)[6]— can use this well-known attack vector to enable attacks which are both expressive and increasingly hard to detect. The fundamental problem is insufficient bounds checking in memory-unsafe languages such as C / C++. Approaches for hardening memory-unsafe programs have been proposed, but tend to incur high performance overheads, and are therefore impractical to deploy [19]. An exception is a technique called *stack canaries* [2], which is both efficient and can detect large classes of attacks. Consequently stack canaries are widely supported by compilers and used in all major operating systems today [2, 9].

Widely deployed stack canary implementations suffer from one or more of the following weaknesses: they 1) rely on canary values that are fixed for a given run of a program [2]; 2) store the reference canary in insecure memory, where an attacker can read or overwrite it [9]; or 3) use only a single canary per stack frame and therefore cannot detect overflows that corrupt only local variables.

The recently introduced ARMv8.3-A pointer authentication (PA) [1] hardware can be used to verify return addresses [14], effectively turning the return address itself into a stack canary. However, PA on its own is susceptible to *reuse attacks*, where an attacker substitutes one authenticated pointer with another [8]. State-of-the-art schemes harden PA return-address protection to ensure that protected return address as statistically unique to a particular control-flow path, and therefore cannot be substituted by an attacker [7].

We propose fine-grained PA-based canaries that: 1) protect individual variables from buffer overflow, 2) do not require secure storage for reference canaries, 3) can use existing return-address protection [7] as an *anchor* to produce canaries which are statistically unique to a particular function call, and 4) are efficient, since they can leverage hardware PA instructions both for canary generation and verification.

Our contributions are:

- **PCan**: A **fine-grained**, and **efficient** PA-based canary scheme (Section 5).
- A **realization of PCan** for LLVM (Section 6).
- **Evaluation** showing that PCan is more secure than existing stack canaries and has only a small performance impact (Section 7).

2 Background

A *stack canary*[2] is a value placed on the stack such that a buffer overflow will overwrite it before corrupting the return

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SysTEX '19, October 27, 2019, Huntsville, ON, Canada

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6888-9/19/10…$15.00
https://doi.org/10.1145/3342559.3365336

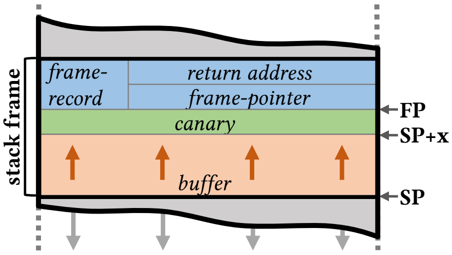


Figure 1. A stack canary is a value placed on the stack so that it will be overwritten by a stack buffer that overflows to the return address. It allows detection of overflows by verifying the integrity of the canary before function return.

address (Figure 1). The buffer overflow can be detected by verifying the integrity of the canary before return.

The initially proposed canaries were randomly generated 32-bit values assigned at process startup and stored within the process memory [2]. The canaries must remain confidential to prevent an attacker $\mathcal{A}$ from avoiding detection by writing back the correct canary when triggering the buffer overflow. *Terminator canaries* [3], consisting of string terminator values (e.g., `0x0`, `EOF`, and `0xFF`) can prevent $\mathcal{A}$ from using string operations to read or write whole canaries, thereby thwarting run-time canary harvesting. Another approach is to re-generate canaries at run-time, for instance by masking them with the return address [4]. However, such techniques rely on the secrecy of the masking value.

Multi-threaded and forked processes can be vulnerable to guessing attacks if the canaries are shared. If the child process or thread is restarted after a crash, $\mathcal{A}$ can execute a large number of guesses without resetting the canary. Moreover, such attacks can utilize incremental—e.g., byte-for-byte—guessing to efficiently find the canary value [10].

Strong adversaries with arbitrary memory read or write access can trivially circumvent any canary based solution; using reads alone allows $\mathcal{A}$ to first read the correct canaries from memory and then perform a sequential overwrite that writes back the correct canaries while corrupting other data.

2.1 Stack canaries in modern compilers

Modern compilers such as LLVM/Clang and GCC provide the `-fstack-protector` feature that can detect stack-buffer overflows[1]. It is primarily designed to detect stack overflows that occur in string manipulation. The default `-fstack-protector` option includes a canary only when a function defines a character array that is larger than a particular threshold. The default threshold value in GCC and LLVM is 8 characters, but in practice the threshold is often lowered to 4 to provide better coverage. However, a stack overflow can occur on other types of variables. The `-fstack-protector-all` option adds a canary to **all** functions. However, it can incur a substantial use of stack space and run-time overhead in complex programs.

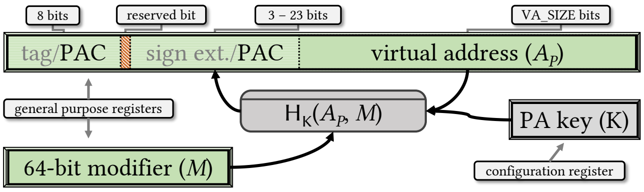


Figure 2. PA verifies pointers using an embedded PAC generated from a pointer's address, a 64-bit modifier and a hardware-protected key. (Figure from [7])

The `-fstack-protector-strong` option provides a better trade-off between function coverage, run-time performance, and memory cost of stack canaries. It adds a canary to any function that 1) uses a local variable's address as part of the right-hand side of an assignment or function argument, 2) includes a local variable that is an array, regardless of the array type or length, and 3) uses register-local variables.

Today `-fstack-protector-strong` is enabled by default for user-space applications in major Linux distributions, such as Debian and its derivatives[2]. The `-fstack-protector` protects non-overflowing variables by rearranging the stack such that an overflow cannot corrupt them; but this protection cannot protect other buffers. On AArch64 the LLVM/-Clang implementation of `-fstack-protector` uses a single reference canary value for the whole program. This in-memory reference canary is used to both store and verify the stack canary on function entry and return, respectively.

2.2 ARMv8-A Pointer Authentication

ARMv8.3-A PA is a instruction set architecture (ISA) extension that allows efficient generation and verification of pointer authentication codes (PACs); i.e., keyed message authentication codes (MACs) based on a pointer's address and a 64-bit modifier [1]. The PAC is embedded in the unused bits of a pointer (Figure 2). On 64-bit ARM, the default Linux configuration uses 16-bit PACs. GNU/Linux has since 5.0 provided support for using PA in user-space. PA provides new instructions for generating and verifying PACs in pointers, and a generic `pacga` instruction for constructing a 32-bit MAC based on two 64-bit input registers. After a PAC is added to a pointer, e.g., using the `pacia` instruction, it can be verified later using the corresponding authentication instruction, in this case `autia`. A failed verification does not immediately cause an exception. Instead, PA corrupts the pointer so that any subsequent instruction fetch or dereference based on it causes a memory translation fault. The `pacga` instruction is an exception as it outputs the produced

[1] https://lists.llvm.org/pipermail/cfe-dev/2017-April/053662.html

[2] https://wiki.debian.org/Hardening

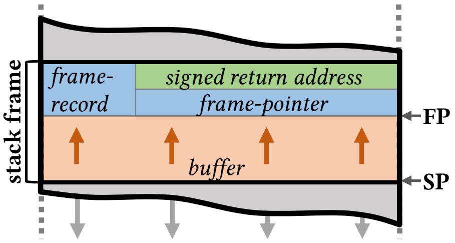


Figure 3. The signed return address generated by `-msign-return-address` effectively serves as a canary by allowing detection of stack-buffer overflows.

PAC to a given destination register; verification in this case must be performed manually by comparing register values.

Current versions of GCC and LLVM/Clang provide the `-msign-return-address` feature that uses PA to protect return addresses [14]. It signs the return address with the stack pointer (SP) as modifier using `pacia lr, sp`. The integrity of the return address is verified before return by issuing the corresponding authentication instruction `autia lr, sp`. Signed return addresses provides similar protection to stack canaries, i.e., if a stack-buffer overflow corrupts the return address, this is detected when the return address is verified before returning from a function (Figure 3). However, PA is vulnerable to reuse attacks where previously encountered signed pointers can be used to used to replace latter signed pointers using the same key and modifier [8]. For instance, `-msign-return-address` can be circumvented by reusing a prior return address signed using the same SP value.

3 Adversary Model

In this work we consider an adversary $\mathcal{A}$ that attempts to compromise the memory safety of a user-space process by exploiting a stack-buffer overflow. We do not consider adversaries at kernel or higher privilege levels. However, $\mathcal{A}$ can: 1) trigger any existing stack-buffer overflow, 2) use stack-buffer over-reads to read memory and 3) repeatedly restart the process and any child processes or threads in an attempt to brute force canaries. Adversaries with arbitrary memory read or write access cannot be thwarted with canary-based approaches and are beyond the scope of this work.

We assume that $\mathcal{A}$ can analyze the target binary and therefore knows the exact stack layout of functions (barring variable-length buffers). This enables $\mathcal{A}$ to target individual local variables reachable from a particular buffer overflow without overflowing canaries past the local variables (Figure 4). If $\mathcal{A}$ further manages to exploit a buffer over-read or other memory disclosure vulnerability, they could also overflow past the canary by simply replacing the correct canary value during the overflow.

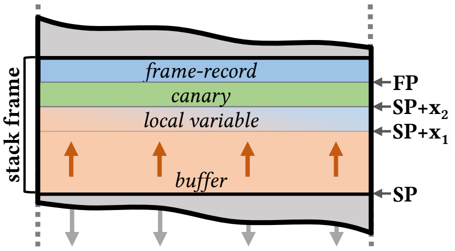


Figure 4. When using only one canary per stack frame, an attacker could overflow a stack buffer to corrupt local variables without overwriting the canary.

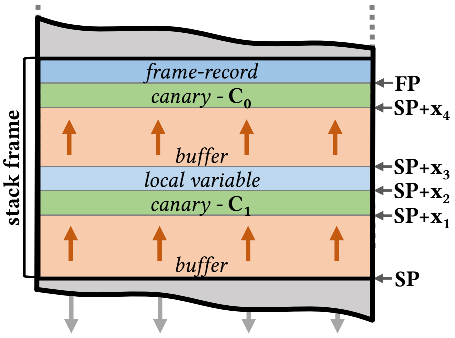


Figure 5. To detect all overflows we we inject canaries after any vulnerable stack buffer.

4 Requirements

To detect linear buffer overflows we require a design that fulfills the following requirements:

R1 Each canary value should be statistically unique.
R2 Reference canaries must not be modifiable by $\mathcal{A}$.
R3 A stack-buffer overflow must always corrupt a canary.

5 Design

We propose PCan, a PA-based canary design that employs multiple function-specific canaries. By placing canaries after any array that could overflow (Figure 5), PCan can detect overflows that only corrupt local variables. This prevents $\mathcal{A}$ from performing precise overflows that corrupt only local variables without overwriting the canary (**R3**). To exploit an overflow without detection $\mathcal{A}$ is instead forced to learn the correct canary values and write them back into place.

In contrast to traditional approaches, PCan avoids exposing reference canaries in memory (**R2**). Instead, canaries are re-generated or verified directly using PA. $\mathcal{A}$ thus cannot manipulate the reference canaries, and must instead leak the specific on-stack canary or attempt a brute-force attack.

The canaries are generated with PA, using a modifier m consisting of a 16-bit function identifier and the least-significant 48 bits from SP: $m = \mathsf{SP} * 2^{16} + \texttt{function-id}$. This modifier makes canaries function-dependent and, when SP differs, different for each call to the same function (**R1**). Because the canaries are generated at run-time and the PA keys are randomly set on each execution, the generated canaries are also statistically unique for each execution. To avoid detection $\mathcal{A}$ must acquire the exact stack-canary belonging to the specific function call and cannot rely on pre-calculated canaries or those belonging to other function calls.

5.1 PA-based canaries

PA-based return-address protection [7, 8, 14] already effectively serves as a canary by detecting return-address corruption. We therefore propose a design that can be efficiently and easily integrated with existing return-address protection schemes, but also provide a stand-alone setup. The first canary in a function's stack frame, protecting the return-address, is either a pacga-generated stand-alone canary or the signed return address:

$$C_0 = \begin{cases} \texttt{pacga}(\mathsf{SP}, m) & \text{if stand-alone} \\ \texttt{signed\_return\_address} & \text{if combined} \end{cases}$$

We denote a canary loaded from the stack with C' to indicate that it might have been corrupted by $\mathcal{A}$. Verification of C'_0 is done either by re-generating the stand-alone pacga canary or by relying on the return-address protection to verify it. To verify using pacga we re-generate C_0 and check that $C_0 = C'_0$.

Subsequent canaries, $C_i, i > 0$, consist of signed pointers to the previous canary:

$$C_i = \mathsf{pacda}(Cptr_{i-1}, m), i > 0$$

where $Cptr_i$ is a pointer to C_i. Verification of $C'_i, i > 0$ is done by authenticating and loading the canary to retrieve C'_{i-1}. If any C'_i is corrupted, authentication fails, causing the subsequent load to fault (Section 2.2) A successful chain of loads will yield C_0, which is then verified as detailed above.

Our stand-alone scheme is more powerful than `-msign-return-address` in that it does not rely solely on the SP value. However, other schemes might provide better protection for the return address. For instance, PACStack [7] proposes a scheme that uses statistically unique modifiers to protect return addresses by maintaining the head of a chain of PACs in a single reserved register. We propose that PCan could be combined with such a mechanism by defining m as the PACStack authentication token $auth_i$ and C_0 as the PACStack protected return address. Because the m in this case would be statistically unique to a specific call-flow this would also harden the canaries C_i for $i > 0$.

6 Implementation

We implement PCan as an extension to LLVM 8.0 and using the stand-alone pacga approach (Section 5.1). To instrument the LLVM Intermediate Representation (IR) we added new LLVM intrinsics for generating and verifying PCan canaries. These intrinsics, along with instructions for storing and loading the canaries, are added through IR transformations before entering the target-specific compiler backend. We define corresponding target-specific intrinsics to leverage built-in register allocation before converting the intrinsics to hardware instructions in the pre-emit stage.

```
canary-creation:
  mov   x8, sp
  movk  x8, #3, lsl #48  ; x8 ← mod
  pacga x10, sp, x8      ; x10 ← $C_0$
  sub   x9, x29, #0x8    ; x9 ← $Cptr_0$
  pacda x9, x10          ; x9 ← $C_1$
  str   x9, [sp, #40]    ; store $C_1$
  stur  x10, [x29, #-8]  ; store $C_0$
```

Listing 1. For a function with two vulnerable stack buffers PCan generates and stores two canaries.

```
canary-verification:
  ldr   x8, [sp, #40]    ; $x8$ ← $C'_1$
  mov   x8, sp
  movk  x8, #3, lsl #48  ; x8 ← mod
  autda x8, x29          ; authenticate $C'_1$
  ldr   x8, [x8]         ; $x8$ ← $C'_0$
  pacga x9, sp, x9       ; $x9$ ← $C_0$
  cmp   x8, x9           ; check $C_0 = C'_0$
```

Listing 2. To verify the integrity of canaries PCan first loads C'_1, then authenticates it before using it to load C'_0, which in turn is compared to the re-generated C_0.

6.1 Canary creation

To instrument the function prologue PCan locates LLVM `alloca` instructions that allocate buffers in the `entry` basic block of each function. A new 64-bit allocation for the canaries is added after each existing `alloca`. Intrinsics for generating the canaries and storing them are then added. The instrumented code will generate a larger stack-frame to accommodate the canaries and include code to generate and store the canary values (Listing 1).

6.2 Canary verification

To verify canaries in the function epilogue, PCan loads them in reverse order, starting from the last C_n (Listing 2). Each canary C'_i is authenticated using autda and then dereferenced to acquire the next canary C'_{i-1}. To verify the final canary, C'_0, PCan first re-generates C_0 and then performs a value comparison. Upon failure, an error handler is invoked, otherwise the function is allowed to return normally. As suggested in Section 5.1, the final canary can be replaced with a return-address protection scheme. The return address then serves as a canary that is verified using the corresponding protection scheme (e.g., `-msign-return-address`).

7 Evaluation

Due to lack of publicly available PA-capable hardware we have used an evaluation approach similar to prior work [7, 8].

benchmark	stack-protector		PCan	
505.mcf_r	−4.78%	(4.55)	−0.05%	(0.13)
519.lbm_r	−0.01%	(0.01)	0.04%	(0.02)
525.x264_r	−0.01%	(0.01)	1.80%	(0.01)
538.imagick_r	−0.01%	(0.01)	0.19%	(0.01)
544.nab_r	0.05%	(0.24)	−0.18%	(0.16)
557.xz_r	0.00%	(0.03)	0.04%	(0.06)
geo.mean.	−0.08%		0.03%	

Table 1. SPEC CPU 2017 performance overhead of PCan and `-fstack-protector-strong` compared to an uninstrumented baseline (standard error in parenthesis). Results indicate that both schemes incur a negligible overhead (geometric mean of 0.3% and < 0%, respectively).

We used the ARMv8-A *Base Platform Fixed Virtual Platform (FVP), based on Fast Models 11.5*, which supports ARMv8.3-A for functional evaluation. For performance evaluation we used the PA-analogue from prior work [7] and performed measurements on a 96board Kirin 620 HiKey (LeMaker version) with an ARMv8-A Cortex A53 Octa-core CPU (1.2GHz) / 2GB LPDDR3 SDRAM (800MHz) / 8GB eMMC, running the Linux kernel v4.18.0 and BusyBox v1.29.2.

7.1 Performance

We evaluated the performance of PCan using the SPEC CPU 2017[3] benchmark, and running it on the HiKey board. We cross-compiled the benchmarks on an x86 system using whole program LLVM[4], and timed the execution of the individual benchmark programs using the `time` utility. Results are reported normalized to a baseline measured without PCan instrumentation and compiled with `-fno-stack-protector` (Table 1). We compare this baseline to two different setups; one using only `-fstack-protector-strong` and another using `-fno-stack-protector` and PCan instrumentation. Our results indicate that PCan incurs a very low overhead with a geometric mean of 0.30%. In some cases `-fstack-protector-strong` caused the benchmarks, we suspect this is caused by it rearranging the stack. Measurements were repeated 20 times and all binaries were compiled with `-O2` optimizations enabled.

7.2 Security

The initial `pacga` canaries used by PCan provide similar security to traditional canaries. To perform an overflow while avoiding detection $\mathcal{A}$ must achieve the following goals: 1) find the location of canaries in relation to the overflown buffer, 2) leak the specific canary values on the stack, and 3) write back the correct canaries when performing the buffer overflow. In our adversary model step 1) is trivial; $\mathcal{A}$ can inspect the binary to analyze the stack layout. Step 2) could be achieved by leaking or modifying the in-memory reference values, but because PCan generates canaries on-demand, $\mathcal{A}$ is forced to leak the values from the stack (**R2**). Moreover, because the canaries are statistically unique to a function and SP value $\mathcal{A}$ cannot rely on finding just any canary and substitute it with one in the overflown stack frame (**R1**). This limits the scope of attacks, as both the memory leak and overflow must happen within the lifetime of the attacked stack frame. By using multiple canaries—one after each buffer—PCan can detect overflows that only touch local variables (**R3**). Based on our evaluation PCan thus provides comprehensive protection with an overhead similar to currently deployed defenses.

8 Related Work

After the seminal article "Smashing the Stack for fun and profit" [12], the notion of *canaries* as a protection against buffer overflow was first introduced in StackGuard [2], and initial GCC compiler support appeared at the same time. StackGuard proposes to use a random canary, stored at the top of the stack (or in the thread local storage memory area), during program launch to thwart canary harvesting from the compiled code. The threat of canary harvesting and the added protection (especially for C) provided by terminator canaries was identified shortly thereafter [3]. The problem of canary copy and re-use was already identified by Etoh and Yoda in 2000 [4], where the stack-frame based canary protection was augmented by masking the canary value with the function return address. Later, Strackx et al. [18] argue against the futility of storing secrets in program memory, which supports using PA to generate canaries dynamically.

Another shortcoming of canary integrity are cases when the canary mechanism is subject to brute-force attacks, e.g., in the context of process forking. $\mathcal{A}$ could use the canaries in forked child processes as oracles to perform brute-force guessing of canary values. Published solutions against this form of attack includes DynaGuard [13] and DCR [5]. Both solutions keep track of canary positions in the code, and re-initialize all canaries in a child process, at considerable performance overhead. DCR optimizes the canary location tracking by chaining canaries using embedded offsets - we inherit this notion of chaining canaries from their work, although we deploy these for canary validation whereas DCR uses the mechanism for canary rewriting. By combining the SP in the canaries PCan mitigates such attacks, but full protection would would require a similar approach of re-initializing canaries on fork. Finally, the polymorphic canaries by Wang et al. [20] optimize away the need to rewrite canaries during fork, by adding a function-specific random

[3] https://www.spec.org/cpu2017/

[4] https://github.com/travitch/whole-program-llvm

mask to the stack canary, which effectively removes the opportunity for systematic canary brute-forcing.

Heap protection with canaries has received much less attention than stack protection, possibly because the optimal balance between validation and performance overhead when canaries are applied to the heap remains an open problem. The first paper on the subject was Robertson et al. in 2003 [15], but a more recent mechanism — HeapSentry by Nikiforakis et al. [11] puts effort on the unpredictability (randomness) of the heap canaries. HeapSentry consists of a wrapper for the allocator and a kernel module, and exhibits overheads at around 12%. Pointer bounds checking schemes offer protections stronger than canaries alone, but in comparison incur significant performance overheads [19].

9 Future Work

Our current approach only protects stack-based variables with a static size. Canaries for dynamic allocations cannot be verified in the prologue because they might be either out of scope or overwritten by later dynamic allocations, and are currently not used by PCan. To prevent attacks that corrupt dynamic allocations, we propose to add instrumentation that protects dynamic allocations based on their life-time, i.e., which verifies the associated canaries immediately when the allocation goes out of scope. The existing LLVM allocation life-time tracking could be leveraged to implement this addition without significant changes to the compiler.

We plan to refine and expand our canary approach by using compile-time analysis—i.e., the StackSafetyAnalysis [5] of LLVM—to omit instrumentation of buffers that can be statically shown to be safe. In some cases $\mathcal{A}$ could achieve their goal before function return, i.e., before the canary corruption is detected. Such attacks could be detected earlier by utilizing the StackSafetyAnalysis to add checks after vulnerable steps during function execution, before the function epilogue.

We also plan to extend PCan instrumentation to cover heap allocations, similar to HeapSentry [15]. Because the PA-keys are managed by the kernel, PCan could be used for HeapSentry-like consistency checks from within the kernel, e.g., before executing system-calls.

10 Conclusion

Canaries are a well-established protection mechanism against errors in memory-unsafe programs. We present PCan, which provides hardware-assisted integrity-protection for canaries, inhibiting the most prevalent canary-circumvention techniques. Furthermore, we propose the notion of fine-grained canaries, where canaries not only protect the return address, but also detect overflows in individual data objects. We make available our compiler prototype at https://github.com/pointer-authentication/PCan-llvm, and measure its performance impact. Finally we point out further optimizations for fine-grained canaries, as well a solution path for protecting dynamic allocations.

11 Acknowledgments

This work was supported in part by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the Academy of Finland under grant nr. 309994 (SELIoT), and a Google ASPIRE award.

References

[1] ARM Ltd. 2017. ARMv8 Architecture Reference Manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[2] Crispin Cowan et al. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. USENIX Security '98*. 63–78.

[3] Crispin Cowan et al. 1999. Protecting systems from stack smashing attacks with StackGuard. *Linux Expo* (1999).

[4] Hiroaki Etoh and Kunikazu Yoda. 2000. *Protecting from stack-smashing attacks*. Technical Report. IBM Research Division, Tokyo Research Laboratory.

[5] William H Hawkins et al. 2016. Dynamic canary randomization for improved software security. In *Proc. ACM CISR '16*. 9:1–9:7.

[6] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proc. IEEE S&P '16*. 969–986.

[7] Hans Liljestrand et al. 2019. Authenticated Call Stack. In *Proc. ACM/EDAC/IEEE DAC'19*. Article 223.

[8] Hans Liljestrand et al. 2019. PAC it up: towards pointer integrity using ARM pointer authentication. In *Proc. USENIX Security '19*. 177–194.

[9] David Litchfield. 2003. Defeating the stack based buffer overflow preventation mechanism of Microsoft Windows 2003 Server. In *Black Hat Asia '03*.

[10] Hector Marco-Gisbert and Ismael Ripoll. 2013. Preventing brute force attacks against stack canary protection on networking servers. In *Proc. IEEE NCA '13*. 243–250.

[11] Nick Nikiforakis et al. 2013. HeapSentry: kernel-assisted protection against heap overflows. In *Proc. DIMVA 13'*. 177–196.

[12] Elias Levy (Aleph One). 1996. Smashing the stack for fun and profit. *Phrack* 7, 19 (1996). http://phrack.org/issues/49/14.html

[13] Theofilos Petsios et al. 2015. Dynaguard: Armoring canary-based protections against brute-force attacks. In *Proc. ACM ACSAC '15*. 351–360.

[14] Qualcomm. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[15] William K Robertson et al. 2003. Run-time detection of heap-based overflows. In *Proc. USENIX LISA '03*. 51–60.

[16] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM CCS '07*. 552–561.

[17] Solar Designer. 1997. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[18] Raoul Strackx et al. 2009. Breaking the memory secrecy assumption. In *Proc. ACM EuroSec '09*. 1–8.

[19] László Szekeres et al. 2013. SoK: Eternal war in memory. In *Proc. IEEE S&P '13*. 48–62.

[20] Zhilong Wang et al. 2018. To detect stack buffer overflow with polymorphic canaries. In *Proc. IEEE/IFIP DSN '18*. IEEE, 243–254.

[5] https://llvm.org/docs/StackSafetyAnalysis.html

Publication V

Hans Liljestrand, Thomas Nyman, Lachlan Gunn, Jan-Erik Ekberg, N. Asokan. PACStack: an Authenticated Call Stack. *Submitted*, 20 pages, August 2019.

PACStack: an Authenticated Call Stack

Hans Liljestrand
Aalto University, Finland
Huawei Technologies Oy, Finland
hans.liljestrand@aalto.fi

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Lachlan J. Gunn
Aalto University, Finland
lachlan@gunn.ee

Jan-Erik Ekberg
Huawei Technologies Oy, Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
University of Waterloo, Canada
asokan@acm.org

ABSTRACT

A popular run-time attack technique is to compromise the control-flow integrity of a program by modifying function return addresses on the stack. So far, shadow stacks have proven to be essential for *comprehensively preventing* return address manipulation. Shadow stacks record return addresses in integrity-protected memory secured with hardware-assistance or software access control. Software shadow stacks incur high overheads or trade off security for efficiency. Hardware-assisted shadow stacks are efficient and secure, but require the deployment of special-purpose hardware.

We present *authenticated call stack* (ACS), an approach that uses chained message authentication codes (MACs) to achieve comparable security *without requiring additional hardware support*. We present PACStack, a realization of ACS on the ARMv8.3-A architecture, using its general purpose hardware mechanism for pointer authentication (PA). Via a rigorous security analysis, we show that PACStack achieves security comparable to hardware-assisted shadow stacks without requiring dedicated hardware. We demonstrate that PACStack's performance overhead is negligible (<1%).

1 INTRODUCTION

Traditional code-injection attacks are ineffective in the presence of W⊕X policies that prevent the modification of executable memory [50]. However, code-reuse attacks can alter the run-time behavior of a program without modifying any of its executable code sections. Return-oriented programming (ROP) is a prevalent attack technique that corrupts function return addresses to hijack a program's control flow. ROP can be used to achieve Turing-complete computation by chaining together existing code sequences in the victim program. To prevent ROP, return addresses must be protected when stored in memory. At present, the most powerful protection against ROP is using an *integrity-protected shadow stack* that maintains a secure reference copy of each return address [1]. Integrity of the shadow stack is ensured by making it inaccessible to the adversary either by randomizing its location in memory or by using specialized hardware [28]. Recent software-based shadow stacks show reasonable performance [11], but are vulnerable to an adversary capable of exploiting memory vulnerabilities to infer the location of the shadow stack. To date, only hardware-assisted schemes, such as Intel CET [28], achieve negligible overhead without any security trade-offs. But employing such a custom hardware mechanism incurs a development and deployment cost.

Recent ARM processors include support for general-purposepointer authentication (PA); a hardware extension that uses tweakable message authentication codes (MACs) to sign and verify pointers [2]. One initial use case of PA is the authentication of return addresses [47]. However, current PA schemes are vulnerable to *reuse attacks*, where the adversary can reuse previously observed valid protected pointers [35]. Prior work [35, 47] and current implementations by GCC[1] and LLVM[2] mitigate reuse attacks, but cannot completely prevent them.

In this paper, we propose a new approach, *authenticated call stack* (ACS), providing security comparable to hardware-assisted shadow stacks, with minimal overhead and without requiring new hardware-protected memory. ACS binds all return addresses into a chain of MACs that allow verification of return addresses before their use. We show how ACS can be efficiently realized using ARM PA while resisting reuse attacks. The resulting system, PACStack, can withstand strong adversaries with full memory access. Our contributions are:

- ACS, a new approach for **precise verification of function return addresses** by chaining MACs (Section 5).
- PACStack, a LLVM-based realization of ACS using ARM PA **without requiring additional hardware** (Section 6).
- A systematic evaluation of PACStack security, showing that its **security is comparable to shadow stacks** (Section 7).
- Demonstrating that the **performance overhead** of PACStack **is negligible** (<1%) (Section 8).

For realizing PACStack, we implemented an efficient authenticated stack using ARM PA. This approach may be generalizable to other data structures and applications (Section 10.1). We plan to make our PACStack implemenation and associated evaluation code available as open source.

2 BACKGROUND

2.1 ROP on ARM

In ROP, the adversary exploits a memory vulnerability to manipulate return addresses stored on the stack, thereby altering the program's backward-edge control flow. ROP allows Turing-complete attacks by chaining together multiple gadgets, i.e., adversary-chosen sequences of pre-existing program instructions that together perform the desired operations. ARM architectures use the link register

[1] https://gcc.gnu.org/onlinedocs/gcc/AArch64-Function-Attributes.html
[2] https://reviews.llvm.org/D49793

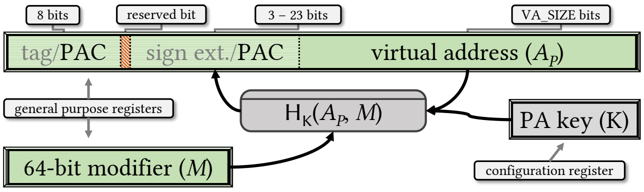


Figure 1: PA uses an embedded authentication token based on the pointer's address, a modifier, and a key.

(LR) to hold the current function's return address. LR is automatically set by the *branch with link* (`bl`) or *branch with link to register* (`blr`) instructions that are used to implement regular and indirect function calls. Because LR is overwritten on call, non-leaf functions must store the return address onto the stack. This opens up the possibility of ROP on ARM architectures [30].

2.2 ARM Pointer Authentication

The ARMv8.3-A PA extension supports calculating and verifying pointer authentication codes (PACs) [2]. A `pac` instruction calculates a keyed tweakable MAC, $\mathsf{H}_K(A_P, M)$, over the address A_P of a pointer P using a 64-bit modifier M as the tweak. The resulting authentication token, referred to as a PAC, is embedded into the unused high-order bits of P. It can be verified using an `aut` instruction that recalculates $\mathsf{H}_K(A_P, M)$, and compares the result to P's PAC.

Since the PAC is stored in unused bits of a pointer, its size is limited by the virtual address size (`VA_SIZE` in Figure 1) and whether address tagging is enabled [2]. On a 64-bit ARM machine running a default Linux kernel, `VA_SIZE` is 39, which leaves 16 bits for the PAC when excluding the reserved and address tag bits. PA provides five different keys; two for code pointers, two for data pointers, and one for generic use. Each key has a separate set of instructions[3], e.g., the `autia` and `pacia` instructions always operate on the instruction key A, stored in the `APIAKey_EL1` register. Access to the key registers and PA configuration registers can be restricted to a higher exception level (EL). Linux v5.0[4] adds full support for PA, such that the kernel (at EL1) manages user-space (EL0) keys and prevents EL0 from modifying them.

As currently specified, PA does not cause a fault on verification failure; instead, it strips the PAC from the pointer P and flips one of the high-order bits such that P becomes invalid. If the invalid pointer is used by an instruction that causes the pointer to be translated, such as load or instruction fetch that dereferences the pointer, the memory management unit issues a memory translation fault.

PA also supports the generic `pacga` instruction, which outputs a 32-bit PAC based on a 64-bit input value and a 64-bit modifier. There is no corresponding verification instruction. To verify the `pacga` PAC, instrumented code must explicitly compare it to the expected value.

2.2.1 PA-based return address protection. Return address protection is the first published PA-based control-flow protection [47]. It is implemented as the `-msign-return-address` feature of GCC and LLVM/Clang.[5] An authenticated return address is computed using `paciasp` (❶ in Listing 1) and verified with `autiasp` (❷ in Listing 1). These instructions implicitly use the value of stack pointer (SP) as the modifier. An adversary cannot create the correct PAC for an arbitrary pointer and therefore cannot modify the return address without causing a fault on function return.

```
epilogue:
  paciasp               ; sign LR                    ❶
  str LR, [SP]          ; push LR onto stack
function_body:
  ...
epilogue:
  ldr LR, [SP]          ; pop stack onto LR
  autiasp               ; verify LR                  ❷
  ret
```

Listing 1: The `-msign-return-address` feature in GCC and LLVM/Clang uses PA to sign and verify the return address in LR when storing and loading it from the stack.

The `-msign-return-address` feature and other prior PA-based solutions are vulnerable to *reuse attacks* where an adversary replaces a valid authenticated return address with another authenticated return address previously read from the process' memory. For a reused PAC to pass verification, both the original and replacement PAC must have been computed using the same PA key and modifier. This applies to any PA scheme, not only authenticated return addresses. For instance, if a constant modifier is used then all pointers based on the same key are interchangeable. Using only the SP value as a modifier reduces the set of interchangeable pointers, but still allows reuse attacks when SP values coincide. Reuse attacks can be mitigated, but not completely prevented, by further narrowing the scope of modifier values [35].

3 ADVERSARY MODEL

In this work, we consider a powerful adversary, $\mathcal{A}$, with arbitrary control of process memory but restricted by a W⊕X policy. Therefore $\mathcal{A}$ can read all process memory, but write operations and execution are restricted such that $\mathcal{A}$ can neither modify program code nor execute memory pages reserved for data (e.g., the program stack). This adversary model is consistent with prior work on run-time attacks [50].

These abilities allow $\mathcal{A}$ to modify any pointer in the process data memory pages. In particular, $\mathcal{A}$ can modify function return addresses while they reside on the program call stack.

In this work, we exclude adversaries with kernel mode privilege escalation capabilities, i.e., $\mathcal{A}$ cannot undermine kernel integrity or confidentiality. As a consequence, $\mathcal{A}$ cannot modify or read sensitive data in kernel memory or kernel-managed registers, such as the PA keys. As in prior work on control-flow integrity (CFI), we do not consider non-control data attacks [13], such as data-oriented programming (DOP) [26].

[3] A full list of PA instructions from [35] is available in Appendix D.

[4] https://kernelnewbies.org/Linux_5.0#ARM_pointer_authentication

[5] https://gcc.gnu.org/gcc-7/changes.html and https://reviews.llvm.org/D49793

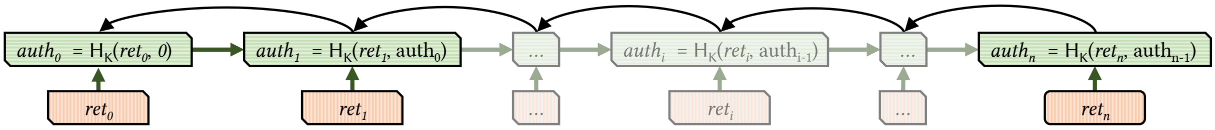

$auth_0 = H_K(ret_0, 0)$
$auth_1 = H_K(ret_1, auth_0)$
…
$auth_i = H_K(ret_i, auth_{i-1})$
…
$auth_n = H_K(ret_n, auth_{n-1})$
ret_0
ret_1
…
ret_i
…
ret_n

Figure 2: ACS is an chained MAC of tokens $auth_i, i \in [0, n-1]$ that are cryptographically bound to the corresponding return addresses, $ret_i, i \in [0, n]$, and the last $auth_n$.

4 REQUIREMENTS & ASSUMPTIONS

Our goal is to thwart $\mathcal{A}$ who modifies function return addresses on the call stack in order to hijack the program control flow. We define the following requirements for our solution:

- **R1** *Return address integrity*: Detect if a function return address has been modified while in program memory.
- **R2** *Memory disclosure tolerance*: Remain effective even when $\mathcal{A}$ can read the entire process address space.
- **R3** *Compatibility*: Be applicable to typical (standards-compliant) C code, without requiring source code modifications.
- **R4** *Performance*: Impose only minimal run-time performance and memory overhead, while meeting **R1**–**R3**.

We make the following assumptions about the system:

- **A1** *A W⊕X policy* that protects code memory pages from modification by non-privileged processes. W⊕X is today supported by all major processor architectures, including ARMv8-A.
- **A2** *Coarse-grained forward-edge CFI.* We assume that ACS is combined with a CFI solution that restricts forward control-flow transfers to a set of valid targets. Specifically, we assume that indirect function-calls always target the beginning of a function and that indirect jumps to arbitrary addresses is infeasible. This property can be satisfied by several pre-existing software-only CFI solutions with reasonable overhead [1, 18, 31, 37], as well as with negligible overhead by using hardware-assisted mechanisms like ARM PA itself [35], branch target indicators [2], or TrustZone-M [5, 40].

Coarse-grained forward-edge CFI (**A2**) and W⊕X (**A1**) are used to prevent $\mathcal{A}$ from tampering with the instrumentation that maintains the ACS, as discussed in Section 7.2.

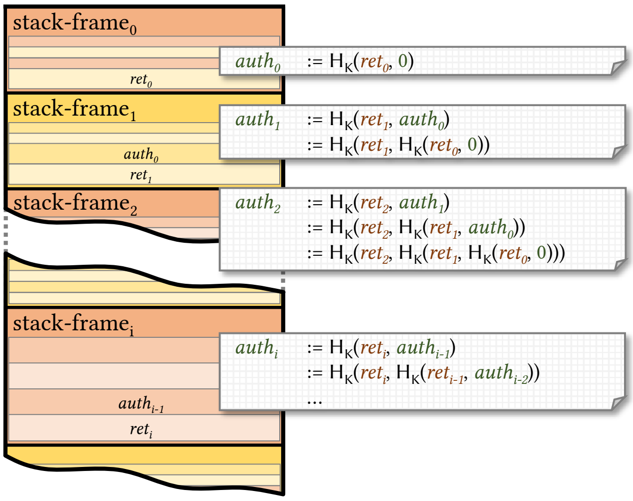

stack-frame$_0$
ret_0
$auth_0 := H_K(ret_0, 0)$
stack-frame$_1$
$auth_0$
ret_1
$auth_1 := H_K(ret_1, auth_0)$
$:= H_K(ret_1, H_K(ret_0, 0))$
stack-frame$_2$
$auth_2 := H_K(ret_2, auth_1)$
$:= H_K(ret_2, H_K(ret_1, auth_0))$
$:= H_K(ret_2, H_K(ret_1, H_K(ret_0, 0)))$
stack-frame$_i$
$auth_{i-1}$
ret_i
$auth_i := H_K(ret_i, auth_{i-1})$
$:= H_K(ret_i, H_K(ret_{i-1}, auth_{i-2}))$
…

Figure 3: ACS stores return addresses and intermediate authentication tokens, $auth_i, i \in [0, n-1]$, on the stack. Only the last token ($auth_n$) needs to be securely stored.

5 DESIGN: AUTHENTICATED CALL STACK

In this section we present our general design for ACS, not tied to a particular hardware-assisted mechanism. In Section 6, we present our implementation that efficiently realizes ACS using PA. While PA approximates pointer integrity it falls short when the modifier is not unique to a pointer. Our key idea is to provide a modifier for the return address by cryptographically binding it to all previous return addresses in the call stack. This makes the modifier statistically unique to a particular control-flow path, thus preventing reuse-type attacks and allowing precise verification of return addresses.

Recall that on ARM systems, the return address is initially stored in LR, which cannot be manipulated by $\mathcal{A}$ (Section 2.1). However, non-leaf functions need to store their return address on the stack before invoking a nested function. The return addresses $ret_i, i \in [0, n-1]$ (where n is the depth of the call stack in terms of active function records) must thus always be stored on the stack, where $\mathcal{A}$ can modify them by exploiting memory vulnerabilities. ACS protects these values by computing a series of *chained* authentication tokens $auth_i, i \in [0, n]$ that cryptographically bind the latest $auth_n$ to all return addresses $ret_i, i \in [0, n-1]$ stored on the stack (Figure 2). Only the MAC key and the last authentication token $auth_n$ must be stored securely to ensure that previous $auth$ tokens and return addresses can be correctly verified when unwinding the call stack. We use a tweakable MAC function H_K to generate a b-bit authentication token $auth_i$:

$$auth_i = \begin{cases} \mathsf{H}_\mathsf{K}(ret_i, auth_{i-1}) & \text{if } i > 0 \\ \mathsf{H}_\mathsf{K}(ret_i, 0) & \text{if } i = 0 \end{cases}$$

$auth_n$ is maintained in a register unmodifiable by $\mathcal{A}$. Figure 3 shows how authentication tokens and return addresses are stored on the call stack. On function calls, $auth_i$ is retained across the call to the callee, which calculates $auth_{i+1}$ and stores both $auth_i$ and the corresponding return address ret_{i+1} on its stack frame. On return, $auth'_{i-1}$ and ret'_i values are loaded from the stack and are verified by comparing $\mathsf{H}_\mathsf{K}(auth'_{i-1}, ret'_i)$ to $auth_i$. If the results differ, then one or both of the loaded values have been corrupted (**R1**). Otherwise, they are valid—i.e., $auth'_{i-1} = auth_{i-1}$ and $ret'_i = ret_i$—in which case $auth_i$ is replaced with the verified $auth_{i-1}$ in the secure register before the function returns to ret_i.

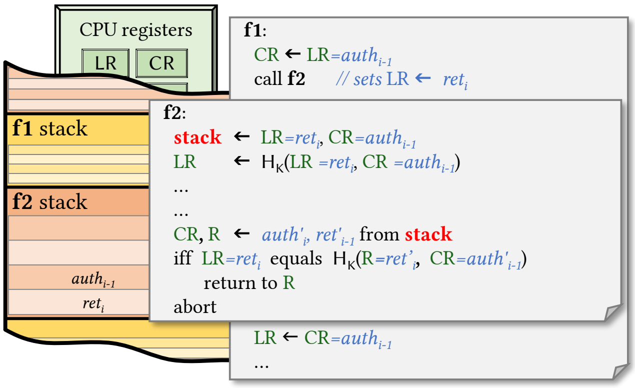

CPU registers
LR
CR
f1 stack
f2 stack
$auth_{i-1}$
ret_i
f1:
CR ← LR=$auth_{i-1}$
call f2 // sets LR ← ret_i
f2:
stack ← LR=ret_i, CR=$auth_{i-1}$
LR ← H_K(LR =ret_i, CR =$auth_{i-1}$)
…
…
CR, R ← $auth'_i$, ret'_{i-1} from stack
iff LR=ret_i equals H_K(R=ret'_i, CR=$auth'_{i-1}$)
return to R
abort
LR ← CR=$auth_{i-1}$
…

Figure 4: To maintain the integrity of ACS the last authentication token is maintained and retained through function calls in the designated CR. The notation x' indicates that x is read from the stack and may have been compromised.

5.1 Authenticated return addresses

We can avoid the need to maintain separate *auth* and *ret* values by defining a combined *authenticated return address*:

$$aret_i = auth_i \parallel ret_i, \text{where}$$

$$auth_i = \begin{cases} \mathsf{H}_K(ret_i, aret_{i-1}) & \text{if } i > 0 \\ \mathsf{H}_K(ret_i, 0) & \text{if } i = 0 \end{cases}$$

We call $auth_i$ and the corresponding $aret_i$ *valid* if they are equal to $\mathsf{H}_K(ret_i, aret_{i-1})$ for some given $aret_{i-1}$.

In this variant, not only the current authentication token, but also the current return address are securely stored. Because the plain return address ret_i is never stored on the stack, $\mathcal{A}$ is limited to manipulating the earlier authenticated return addresses on stack, i.e., $aret_i, i \in [0, n-1]$. A compromised authenticated return address must therefore pass two authentications before use: first when being restored from the stack, and second, when being used as the target of a function return. We discuss the security properties in Section 7.

The remainder of Section 5 will focus on *aret*, but unless otherwise noted, similar properties also apply for separate *auth* tokens.

5.2 Securing the authentication token

The current authenticated return address $aret_n$, is secured by keeping it exclusively in a CPU register. On processors with a dedicated link register, LR can be used to store *aret*; otherwise an additional register must be reserved for this purpose. On function calls, *aret* must be securely retained during a function call that overwrites LR. This is done by modifying the calling convention such that *aret* is kept in a specific register which we call a *chain register* (CR) (Figure 4).

ACS protects the integrity of backward-edge control-flow transfers. Combined with coarse-grained forward-edge CFI (Assumption **A2**), it ensures that: 1) immediately after function return, the $aret_n$ in CR is valid, 2) at function entry the $aret_{n-1}$ stored in CR is valid, and 3) LR is always used as or set to a valid *aret*. This ensures that token updates are done securely, and that the ACS instrumentation cannot be bypassed or used to generate arbitrary authenticated return addresses.

5.3 Mitigation of hash-collisions: authentication token masking

Though $aret_n$ is protected by hardware, the fact that it is embedded in the return pointer means that the size b of the authentication token *auth* is limited by the pointer address size. This is significant, as collisions can be found after $\mathcal{A}$ has seen, on average, approximately $1.253 \cdot 2^{b/2}$ tokens [48, Section 1.4.2] (e.g., 321 tokens for $b = 16$). Despite this, we can still prevent $\mathcal{A}$ from *recognizing* collisions, thus forcing $\mathcal{A}$ to guess which authenticated return addresses yield a collision, succeeding with a probability 2^{-b}. The *auth* of any *aret* stored on the stack is masked using a pseudo-random value derived from the previous *aret* value:

$$auth_i = \mathsf{H}_K(ret_i, aret_{i-1}) \oplus \mathsf{H}_K(0, aret_{i-1}).$$

The mask $\mathsf{H}_K(0, aret_{i-1})$ is exclusive-OR-ed with $\mathsf{H}_K(ret_i, aret_{i-1})$ after it is generated and before it is authenticated, thereby preventing $\mathcal{A}$ from identifying opportunities for pointer reuse.

5.4 Mitigation of brute-force guessing: re-seeding authentication token chain

A brute force attack against PA where $\mathcal{A}$ guesses a PAC correctly succeeds with probability p for a b-bit PAC after $\frac{\log(1-p)}{\log(1-2^{-b})}$ guesses, on the assumption that an authentication attempt of an incorrect PAC terminates the program and subsequent program runs receive a new, random set of PA keys [35] (the current behavior in Linux 5.0).

However, currrently pre-forked or multithreaded programs share the PA key between the parent and sibling processes / threads. This could allow $\mathcal{A}$ to target a vulnerability in a sibling, and unless a failed authentication terminates the entire process tree that shares the PA key, $\mathcal{A}$ can attempt a new guess against another sibling process. In this scenario, 2^{b-1} guesses on average are enough to guess a b-bit PAC [35].

Multi-threaded applications are also affected since address translation errors due to PAC authentication failures are delivered in Linux via the SIGSEGV signal which is always directed agains the offending thread[6], and the thread cannot change the signal's disposition such that it would not be delivered.

Liljestrand et al. [35] recommend hardening pre-forking and multi-threaded applications against guessing attacks by having the application restart all of its processes if the number of PAC failures in child processes exceeds a pre-defined threshold. Since ACS does not exhibit false positives in a typical program (a corrupt return address is a strong indication that the process is being subject to a run-time attack), we recommend an alternative mitigation specific to ACS: *"re-seeding"* the *auth* calculation after a fork or thread creation. For example, calculating $auth_0 = \mathsf{H}_K(ret_0, pid/tid)$ where the *pid* / *tid* corresponds to the process or thread ID, or any other value unique to the task. This solution is straightforward to apply to threads, as a return from the function starting the thread causes the thread to exit. Therefore, the ACS for the thread stacks can be made disjoint from the main ACS chain. Forked processes may include *auth* tokens generated by the parent process in stack frames inherited from the parent. If a child process never returns to

[6] http://man7.org/linux/man-pages/man7/signal.7.html

```
#include <setjmp.h>

jmp_buf ebuf;

void try_catch() {
  int err;

  if (!(err = setjmp(ebuf))) { // iff ebuf set      ①
    checked_func();             // after setjmp
  } else {
    handle_error(err);          // after longjmp    ②
  }
}

void checked_func() {
  // ...
  longjmp(ebuf, E_NUM); // throw exception          ③
  // ...
}

int main() {
  try_catch();
  longjmp(ebuf, E_NUM); // undefined behavior!      ④
}
```

Listing 2: `setjmp` / `longjmp` allows the programmer to transfer execution to another location, potentially in another function. The location, and the state of the environment after the transfer, is determined by an in-memory buffer containing the calling environment of a previous `setjmp` call. Calling `longjmp` after the calling environment is destroyed results in undefined behavior.

inherited stack frames, re-seeding any new *auth* tokens beyond the point of the fork is sufficient. However, if the child process returns to inherited stack frames, the ACS must be re-seeded starting from $auth_0$ by rewriting any *auth* tokens in pre-existing stack frames; similar to some stack canary re-randomization schemes [24, 45].

5.5 Irregular stack unwinding

The C standard includes the `setjmp` / `longjmp` programming interface, which can be used to add exception-like functionality to C (Listing 2). The `longjmp` C function executes a non-local jump to a prior calling environment stored using the `setjmp` function. At `setjmp`, callee-saved registers (whose values are guaranteed to persist through function invocations), as well as the stack pointer SP and return address are stored in the given `jmp_buf` buffer (① in Listing 2). `setjmp` returns `0` to indicate that execution is continuing directly after the call. Upon executing `longjmp`, the environment is restored from `jmp_buf` (③); program execution continues at the `setjmp` return site with a non-zero value (②).Calling `longjmp` using an expired buffer, i.e., after the corresponding `setjmp` caller has returned (④), results in undefined behavior (the implications of this are discussed in Section 10.2). Because `jmp_buf` also stores the latest authenticated token, ACS needs a mechanism to ensure its integrity when using `setjmp` and `longjmp`.

```
call-site
  mov    X28, LR          ; CR ← auth_i      ❶
  bl     @func            ; LR ← ret_{i+1}   ❷
  mov    LR, X28          ; LR ← auth_i      ❸
```

Listing 3: PACStack retains the last *auth* / *aret* via `CR`, defined as the general purpose register X28.

When stored in memory, the integrity of `jmp_buf` cannot be guaranteed. Nonetheless, the stored $aret_i$ is bound to the corresponding $aret_{i-1}$ on the `setjmp` caller's stack. This ensures that `longjmp` always restores a valid ACS state. To limit the set of values $\mathcal{A}$ can inject into `jmp_buf`, we replace the `setjmp` return address ret_b in `jmp_buf` with $aret_b$, defined as:

$$aret_b = (\mathsf{H}_\mathsf{K}(ret_b, aret_i) \parallel ret_b) \oplus \mathsf{H}_\mathsf{K}(\mathsf{SP}_b, aret_i),$$

where SP_b is the SP value stored in `jmp_buf`. When executing `longjmp`, $aret_b$ is recalculated based on the buffer values to verify that the stored $aret_i$ was stored by a `setjmp`. $\mathcal{A}$ cannot generate the $aret_b$ value for an arbitrary $aret_i$, nor replace $aret_b$ with a previously observed $aret_i$. However, because `longjmp` explicitly allows jumping to prior states, ACS cannot ensure that the target is the *intended one*, i.e., $\mathcal{A}$ could substitute the correct `jmp_buf` with another. Shadow stacks share a similar limitation [17], and cannot guarantee that the intended state has been reached, only that the return address (and stack pointer) in that state is intact.

6 IMPLEMENTATION: PACSTACK

We present PACStack, an ACS realization using ARMv8.3-A PA. PACStack is based on LLVM 7.0 and integrated into the 64-bit ARM backend, used via `llc`, the LLVM static compiler. PACStack adds two compilation passes: 1) to instrument function calls for *aret* propagation, and 2) to instrument function prologues and epilogues. The instrumentation is applied by passing the `-pafss-ng` flag to `llc` when transforming LLVM bitcode to target-specific assembly. We plan to add PACStack support to Clang Compiler source code is available at https://pacstack.github.io

6.1 Function call instrumentation

Recall from Section 5 that ACS can be implemented using separate *auth* and *ret* tokens (variant 1), or using a combined authenticated return address (variant 2).

In both PACStack variants, we designate the general purpose register X28 as the chain register (CR) and reserve it for instrumentation use. PACStack instruments call sites to move *auth* (variant 1) or *aret* (variant 2) to CR (❶ in Listing 3) in order to retain its value through function calls that overwrite link register (LR) (❷). After function return the contents of CR are restored to LR (❸).

The advantage of using X28 is that it is a callee-saved register. Whenever a function uses a callee-saved register, it must also ensure that the old value is restored before return. By using X28 as CR, PACStack can be transparently mixed with uninstrumented code (either PACStack-instrumented applications using uninstrumented libraries, or vice-versa). We discuss the security implications of mixing instrumented and uninstrumented code in Section 10.3.

```
prologue:
  stp    X28, LR, [SP]  ; stack ← auth_{i−1}, ret_i
  pacga  LR, LR, X28    ; LR ← auth_i                              ❶
function_body:
  ...
epilogue:
  ldp    X28, Xr, [SP]  ; CR, Xr ← auth′_{i−1}, ret′_i from stack❷
  pacga  Xd, Xr, X28    ; Xd ← auth′_i                             ❸
  cmp    Xd, LR         ; if (auth′_i ≠ auth_i)                    ❹
  jnz    abort          ;   then abort
  ret    Xr             ; return via Xr to ret_i
```

Listing 4: Variant 1 of PACStack generates and verifies *auth* tokens using pacga (❶ and ❸). Both $auth_{i-1}$ and ret_i are stored on the stack, and are hence validated against $auth_i$ on function return (❷). Where possible, the *store pair* (stp) / *load pair* instructions (ldp) are used to minimize the latency for successive loads / stores.

```
prologue:
  str     X28, [SP]      ; stack ← aret_{i−1}                  ①
  pacib   LR, X28        ; LR ← aret_i                         ②
function_body:
  ...
epilogue:
  ldr     X28, [SP]      ; CR ← aret′_{i−1} from stack         ③
  autib   LR, X28        ; LR ← (ret_i or ret*_i)              ④
  ret
```

Listing 5: At function entry, PACStack stores the prior $aret_{i-1}$ on the stack (①) and generates the new $aret_i$ (②). Before return, $aret_{i-1}$ is loaded from the stack (③) and verified against $aret_i$ (④). On verification failure, LR is set to an invalid address ret_i^*, causing a fault on return.

Our current PACStack implementation reserves X28 exclusively for instrumentation use because the LLVM 7.0 implementation prevents LR-use without substantial changes to compiler internals[7]. However, we expect the performance cost to be negligible, as cases where the compiler needs to utilize all callee-saved registers (X19-X29) are infrequent. Note that reserving exclusive use of a register has also been proposed for shadow stacks on the x86 architecture [11], even though x86 has fewer general purpose registers compared to 64-bit ARM processors. Unlike shadow stacks, ACS in general can avoid consuming additional registers by using LR to store *auth* (variant 1, Section 6.2) or *aret* (variant 2, Section 6.2).

6.2 Authenticated return addresses with PA

Variant 1: generating *auth* with pacga. In this variant, we use pacga to generate *auth* tokens:

$$\text{Xd} \leftarrow auth_i = \begin{cases} \text{pacga}(\text{Xd}, \text{LR} = ret_i, \text{CR} = auth_{i-1}) & \text{if } n > 0 \\ \text{pacga}(\text{Xd}, \text{LR} = ret_i, \text{CR} = any) & \text{if } n = 0 \end{cases}$$

To generate and verify authentication tokens, PACStack instruments function prologues and epilogues (Listing 4). In the function prologue, $auth_{i-1}$ and ret_i (in CR and LR, respectively) are stored on the function stack frame and then used to generate a new $auth_i$ with pacga (❶). The $auth_{i-1}$ and ret_i values are then stored on the function stack frame. Before function return, PACStack verifies the $auth'_{i-1}$ and ret'_i read from the stack by calculating the corresponding $auth'_i$ (❸) and comparing it to $auth_i$, stored in LR (❹). For $auth_0$ any value currently in CR is used and stored for later validation. This allows PACStack to operate without explicit initialization by the C Library (libc) startup code. To enable re-seeding the *auth* token chain (Section 5.4), the process and thread initialization, and fork() wrapper in libc should be modified to set the initial value of CR accordingly.

Variant 1 can efficiently compute 32-bit authentication tokens values using pacga. However, it has two drawbacks: First, an additional stack store / load is added for the 4-byte token; to preserve the callee-saved behavior of CR, the full 8-byte register content must be stored on the stack. Second, the output of pacga must be explicitly checked using a comparison and a conditional branch instruction. For this reason, our current implementation only supports variant 2 below. However, in Section 10.1 we discuss using pacga to bind other stack-based write-once data to a specific ACS state.

Variant 2: generating *aret* with autib. In this variant, we use pacib and autib instructions to efficiently calculate and verify ACS authenticated return addresses (Listing 5). These instructions differ from pacga in that the output is an authenticated return address which is directly written to LR:

$$\text{LR} \leftarrow aret_i = \begin{cases} \text{pacib}(\text{LR} = ret_i, \text{CR} = aret_{i-1}) & \text{if } i > 0 \\ \text{pacib}(\text{LR} = ret_i, \text{CR} = any) & \text{if } i = 0 \end{cases}$$

The corresponding verification is similar, and defined as:

$$\text{LR} \leftarrow \text{autib}(\text{LR} = aret_i, \text{CR}) = \begin{cases} ret_i & \text{if } \text{H}_\text{K}(ret_i, \text{CR}) = auth_i \\ ret_i^* & otherwise, \end{cases}$$

where autib will automatically handle verification errors by setting LR to an unusable address ret_i^*. No additional checking is needed; executing a return to ret_i^* causes a address translation fault (Section 2.2). In variant 2, PACStack requires no additional stack space as $aret_{i-1}$ is stored on the stack in place of ret_i, not in addition to it. The value of CR for $aret_0$ is handled identically as in variant 1 for $auth_0$.

6.3 Mitigating hash collisions: PAC masking

To prevent $\mathcal{A}$ from identifying PAC collisions that can be reused to violate the integrity of the call stack, PACStack masks all authentication tokens values before storing them on the stack (Listing 6). A pseudo-random value is obtained by generating a PAC for address 0x0, pacib$(0, aret_{i-1})$ (❶, ❸).

By using pacib we efficiently obtain a pseudo-random value that can be directly applied to the authentication token part of *aret* using only an exclusive-or instruction (eor).

Because this construction uses the same key to generate both authentication tokens and masks, $\mathcal{A}$ must not obtain an $aret_i$ for a ret_i = 0x0 and any existing $aret_{i-1}$. PACStack will never generate such *aret* values, as the return address never points to memory address zero. To prevent leaking the mask directly, it is cleared after

[7] https://github.com/llvm/llvm-project/blob/llvmorg-7.0.0/llvm/lib/Target/AArch64/AArch64CallingConvention.td#L278

```
prologue:
  str    X28, [SP]      ; stack ← aret_{i-1}
  pacib  LR, X28        ; LR ← aret_i^{unmasked}
  mov    Xd, X28        ; Xd ← aret_{i-1}
  mov    X28, #0        ; CR ← 0
  pacib  X28, Xd        ; CR ← mask_i                       ❶
  eor    X28, X28, LR   ; CR ← mask_i ⊕ aret_i^{unmasked}   ❷
function_body:          ;      = aret_i
  ...
epilogue:
  ldr    Xd, [SP]       ; Xd ← aret'_{i-1} from stack
  mov    LR, #0         ; LR ← 0
  pacib  LR, Xd         ; LR ← mask'_i                      ❸
  eor    LR, LR, X28    ; LR ← mask'_i ⊕ aret_i             ❹
  mov    X28, Xd        ; CR ← aret'_{i-1}
  autib  LR, X28
  ret
```

Listing 6: PACStack masks authentication tokens to prevent $\mathcal{A}$ from detecting PAC collisions. The mask is created in CR with $\mathsf{pacib}(0, aret_{i-1})$ (❶), and exclusive-OR-ed with the unmasked authentication token (❷). On return, the mask is recreated (❸) and applied to the masked authentication token $aret_{i-1}$ (❹) before verification.

use. We can thus be certain that no $\mathsf{H_K}(0, x)$ value is visible to $\mathcal{A}$ nor possible to pre-compute without the confidential PA key.

This approach to masking requires two additional PAC calculations for each function activation. Our current implementation supports this as an optional feature that can be invoked using the -pafss-ng-cp flag.

6.4 Irregular stack unwinding

PACStack binds jmp_buf buffers to the $aret_i$ at the time of setjmp call by replacing the setjmp return address ret_b with its authenticated counterpart $aret_b$ (Section 5.5). The libc implementation is not modified; instead setjmp / longjmp calls are replaced with the wrapper functions in Listings 7 and 8.

The setjmp_wrapper wrapper function (Listing 7) executes setjmp and updates the buffer with $aret_b$. PACStack generates $aret_b$ based on the current SP value, CR and the setjmp return address; this avoids the need to read the values setjmp has stored. The longjmp_wrapper (Listing 8) retrieves $aret_b$, $aret_i$, and the SP values from the buffer. It then verifies the values and writes ret_b into jmp_buf.

```
setjmp_wrapper:
  ...                         ; Xb ← jmp_buf
  bl     <setjmp>
ret_b:
  cbz    X0, <return>         ; exit iff after longjmp
  mov    Xd, <ret_b>          ; Xd ← ret_b
  mov    X28, SP;             ; Xd ← SP_b
  pacib  Xd, LR;              ; Xd ← pacib(SP_b, aret_i)
  pacib  X28, LR;             ; CR ← pacib(ret_b, aret_i)
  eor    X28, X28, Xd;        ; CR ← aret_b
  str    X28, [Xb, #r]        ; replace return in jmp_buf
return:
  ...
```

Listing 7: PACStack redirects setjmp calls to its own setjmp_wrapper that binds the return address and $aret_i$ in jmp_buf to the current stack frame and corresponding $aret_{i-1}$. #r is the offset of the return address within jmp_buf.

```
longjmp_wrapper:
  ...                         ; Xb ← jmp_buf
  ldr    X28, [Xb, #a]        ; CR ← aret'_i
  ldr    LR, [Xb, #r]         ; LR ← aret'_b
  ldr    Xd, [Xb, #s]         ; CR ← SP'_b
  autib  Xd, LR;              ; Xd ← autib(SP'_b, aret'_i)
  autib  X28, LR;             ; CR ← autib(aret'_b, aret'_i)
  eor    X28, X28, Xd;        ; CR ← ret_b
  str    X28, [Xb, #r]        ; replace return in jmp_buf
  bl     <longjmp>
  ...
```

Listing 8: Before longjmp, the PACStack longjmp_wrapper checks the binding of the $aret'_b$, ret'_b and sp'_b values stored in jmp_buf. $\mathcal{A}$ cannot generate $aret'_b$ for arbitrary values and therefore cannot inject them in jmp_buf. #r, #a and #s are the offsets to ret_b, CR, and ret_i within jmp_buf.

6.5 Multi-threading

The values of ARMv8-A general purpose registers are stored in memory when entering EL1 (i.e. kernel-mode) from EL0 (i.e. user-mode), for example during context switches and system calls. This must not allow $\mathcal{A}$ to modify the $aret$ values or read the mask, which are both exclusively in either CR or LR during execution (Listings 5 and 6), but must be stored in memory during the context switch. On ARMv8-A, system calls are implemented using the supervisor call instruction (svc) that switches the CPU to EL1 and triggers a configured handler. On 64-bit ARM, Linux v5.0 uses the kernel_entry[8] macro to store all register values on the EL1 stack, where they cannot be accessed by user-space processes. During context switches, callee-saved registers (including CR) and LR are stored in struct cpu_context[9] which belongs to the in-kernel task structure and cannot be accessed by user space. The CR and LR values of a non-executing task are thus securely stored within the kernel, beyond the reach of other processes or other threads within the same process. Thus, no kernel modifications are needed to securely apply PACStack to multi-threaded applications.

7 SECURITY EVALUATION

We address two questions in this section:
1) Is the ACS scheme cryptographically secure?
2) Do ACS's guarantees hold when instantiated as PACStack?

[8] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/entry.S?h=v5.0

[9] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/processor.h?h=v5.0

7.1 ACS security

A generic representation of an attack against ACS is shown in Figure 5. Under normal operation, function C returns to A if called from A (Figure 5a); i.e., when called from A, the return address of C is an address ret_A in A. The goal of $\mathcal{A}$ (Figure 5b) is to cause C to return to some other address ret_B.

Since the authenticated return address $aret_A$ containing ret_A is protected from $\mathcal{A}$, in order to perform a backward-edge control-flow attack, $\mathcal{A}$ must achieve two goals successfully:

AG-Jump: Obtain an authenticated return address $aret_B$, valid with respect to some known modifier, which will validate successfully when C returns.

AG-Load: Violate the integrity of the call stack such that the LR register is loaded with $aret_B$ from AG-Jump rather than the correct authenticated return address $aret_A$.

This requires two returns: one from a 'loader' function to load $\mathcal{A}$'s $aret_B$ into LR, and another from C to the return address ret_B contained in $aret_B$.

In the analyses below, we treat the *auth* token $\mathsf{H}_\mathsf{K}(\mathrm{P}, m)$ as a random oracle with respect to both the pointer P and modifier m. This means that if $\mathsf{H}_\mathsf{K}(P, m)$ has never been computed by a function call, $\mathsf{H}_\mathsf{K}(P, m)$ will match any value with probability 2^{-b}. In the analysis below we assume that programs that share the same PA keys between multiple processes or threads employ the mitigation strategy against brute-force attacks described in Section 5.4. This assumption and the design of ACS ensure that there is no authentication oracle available: the only way to test whether an *auth* token is valid with respect to some address and modifier is to attempt to return using the address and token, triggering a crash if the token is incorrect.

The difficulty of achieving these goals therefore depends on whether $\mathcal{A}$'s desired control-flow violation follows the call graph of the program and whether *auth* tokens are masked. Violating control-flow integrity while still traversing the call graph is easier because this allows $\mathcal{A}$ to harvest *auth* tokens and search for collisions; violations that do not follow the call graph are more difficult because they require that $\mathcal{A}$ make one or more guesses, risking a crash.

7.1.1 Violations that follow the call graph. As $\mathcal{A}$ can harvest authenticated return pointers when they are written to the stack, the short *auth* tokens mean that in the absence of masking an attacker can violate the integrity of the call stack by finding collisions in $\mathsf{H}_\mathsf{K}(\cdot, \cdot)$.

In order to achieve goal AG-Load, $\mathcal{A}$ must find two authenticated return addresses $aret_A$ and $aret_B$, such that i) they are both returned to by a function C, ii) that C contains a call-site to the loader function with a corresponding return address ret_C, and iii) such that

$$\mathsf{H}_\mathsf{K}(ret_C, aret_A) = \mathsf{H}_\mathsf{K}(ret_C, aret_B) = auth_{\text{collision}}. \tag{1}$$

Note that the collisions must be for different values in the second argument only, since that is the value in $\mathcal{A}$'s control. Collisions that require different values for ret_C cannot be exploited because ret_C is in CR and cannot be modified by $\mathcal{A}$.

The *auth* tokens contained in $aret_A$ and $aret_B$ depend on the path that $\mathcal{A}$ has taken through the call graph. $\mathcal{A}$ can obtain as many *auth* tokens with ret_C as a pointer as there are distinct execution paths leading to C. The number of such paths will explode combinatorially as the complexity of the program increases, and cycles in the call graph—as occur in Figure 5—make the number of paths essentially infinite, limited only by available stack space.

Having found such a collision, $\mathcal{A}$ then arranges for function C to be called, traversing the call graph in such a way that it is set up to return to A using $aret_A$. Then, when the function C calls into the loader function, it will set LR to $aret_C$. When the loader function returns to ret_C, it will attempt to load $aret_A$ from the stack. Instead, $\mathcal{A}$ substitutes $aret_B$, which because of (1) will validate correctly when returning to ret_C. Since $aret_B$ is a valid authenticated return address, C will successfully return to ret_B, thereby violating the integrity of the call stack.

More concretely, after collecting q *auth* tokens, according to the birthday paradox [48, Section 1.4.2], the probability that *some* pair collides is:

$$p_{\text{collision}}(q) = 1 - \frac{2^b!}{(2^b - q)! \cdot 2^{q \cdot b}}$$

This quickly approaches 1 as $\mathcal{A}$ collects more tokens, on average occurring after obtaining

$$q = \sqrt{\frac{\pi 2^b}{2}}$$

tokens. With a 16-bit PAC, $\mathcal{A}$ will therefore obtain a collision after harvesting 321 pointers on average.

In order to successfully mount the above attack, $\mathcal{A}$ must find two colliding *auth* tokens and perform the substitution. Without masking, $\mathcal{A}$ can read the *auth* token from the stack. $\mathcal{A}$ can then keep collecting *auth* tokens until they find two that collide; since these are both valid pointers, $\mathcal{A}$ will always succeed once this occurs, thus

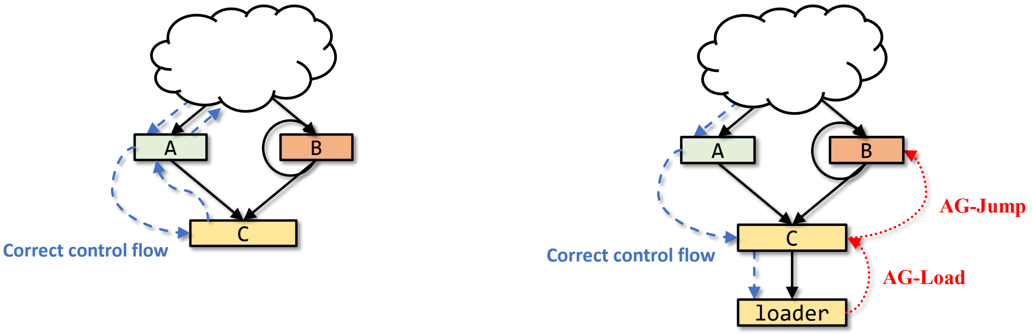
$$\mathbb{P}[\text{AG-Load}|\text{Collision}] = 1.$$

With masking $\mathcal{A}$ cannot identify *auth* token collisions: $aret_A$ and $aret_B$ have different mask values $\mathsf{H}_\mathsf{K}(0, aret_A)$ and $\mathsf{H}_\mathsf{K}(0, aret_B)$. Therefore it is impossible to identify a collision with a probability than by random selection. This means that $\mathcal{A}$ will succeed in the attack above with a probability of 2^{-b}. We give a detailed proof in Appendix A.

In practice, this means that $\mathcal{A}$ can use this attack to traverse the program's call graph, but cannot jump to an address that is not a valid return address for C function.

7.1.2 Violations that leave the call graph. We now consider $\mathcal{A}$'s probability of success when attempting to return to an address ret_B in a way that that *does not* follow the program's call graph.

In this case, the path from B to C has not been traversed, and the instrumentation has never before computed the *auth* token $\mathsf{H}_\mathsf{K}(ret_C, aret_B)$. Therefore, $\mathcal{A}$ succeeds at AG-Load—i.e., $\mathsf{H}_\mathsf{K}(ret_C, aret_B) = \mathsf{H}_\mathsf{K}(ret_C, aret_A)$—with probability $\mathbb{P}[\text{AG-Load}] = 2^{-b}$, irrespective of whether the substituted $aret_B$ is a valid authenticated return address. On failure, which has probability $1 - 2^{-b}$, the process will crash.



(a) Normal control flow. **(b) $\mathcal{A}$'s desired control flow.**

Figure 5: Anatomy of a backward-edge control-flow attack against ACS. In order to force function C to return to B instead of its caller A, $\mathcal{A}$ substitutes their authenticated return address $aret_B$ when some function—the 'loader'—returns to ret_C in function C (goal AG-Load). If $aret_B$ is valid with respect to some known modifier, then at the end of function C the program will return to the corresponding ret_B (goal AG-Jump).

Violation type	No masking	Masking
On-graph	1	2^{-b}
Off-graph to call-site	2^{-b}	2^{-b}
Off-graph to arbitrary address	2^{-2b}	2^{-2b}

Table 1: Maximum probability of success various call-stack integrity violations with and without masking.

$\mathcal{A}$'s probability of then achieving goal AG-Jump depends on whether ret_B is the return address of a valid call-site. If it is, then $\mathcal{A}$ can obtain a valid authenticated return pointer for that location in the same way as in Section 7.1.1, thereby succeeding with probability $\mathbb{P}[\text{AG-Jump}] = 1$. If ret_B has never been used as a return address, then no *auth* token has ever been generated for that pointer. Therefore, AG-Jump is achieved with probability at most $\mathbb{P}[\text{AG-Jump}] = 2^{-b}$; otherwise, the process crashes.

$\mathcal{A}$ can therefore succeed with probability 2^{-b} when the return address is a valid call-site return address, or with probability of 2^{-2b} when the return address is not.

We summarize our results in Table 1.

7.2 Run-time attack resistance of PACStack

PACStack must ensure the integrity of $aret_n$ and the confidentiality of the masks. The former is achieved by storing $aret_n$ in LR or CR, reserved for this purpose, used by regular code, and hence inaccessible to $\mathcal{A}$ (Section 6.1). The latter is maintained as the mask is re-generated each time it is needed, only stored in LR, and cleared after use (Section 6.3). This holds true also in multi-threaded environments (Section 6.5).

Recent results have shown that traditional CFI solutions are unable to withstand control-flow bending [12]; attacks where each control-flow transfer follows the program's CFG, but the program execution trace conforms to no feasible benign execution trace. PACStack—or ACS in general—is not susceptible to backward-edge control-flow bending, because it precisely protects the integrity of the authenticated return addresses while they remain on the stack. $\mathcal{A}$ cannot trick PACStack to deviate from an expected return flow by replacing $aret_n$ with a valid, but outdated $aret$ value, because PACStack never writes $aret_n$ onto the stack. $\mathcal{A}$ also cannot reliably exploit PAC collisions to replace part of the $aret$ chain, as each $aret$ is masked. $\mathcal{A}$ cannot tamper with the instrumentation itself by modifying the instructions in memory (Assumption **A1**). By requiring coarse-grained forward-edge CFI (Assumption **A2**), PACStack ensures that *auth* token calculations and masking are executed atomically and cannot be used to manipulate ret_i, $aret_{i-1}$ or the mask during the function prologue and epilogue. This holds even if the forward-edge CFI is susceptible to control-flow bending.

7.2.1 *Tail-calls and signing gadgets.*

A recent discovery by Google Project Zero[10] shows that PA schemes can be vulnerable to an attack whereby specific code sequences can be used as gadgets to generate PACs for arbitrary pointers. Recall that on PAC verification failure an aut instruction removes the PAC, but corrupts a well-known high-order bit such that the pointer becomes invalid. If a pac instruction adds a PAC to a pointer P with corrupt high-order bits, it treats the high-order bits *as though they were correct* when calculating the new PAC, and flips a well-known bit p of the PAC *if* any high-order bit was corrupt. This means that instruction sequences such as the one shown in Listing 9, consisting of an aut instruction followed by a pac instruction, can be used generate a valid PAC for a pointer even if the original pointer is not valid to begin with. $\mathcal{A}$ writes an arbitrary pointer P to memory (❶) and allows it to be verified. When verification fails, autia removes the PAC, and corrupts the high-order bit in P, writing the resulting P^* to the destination register (❷). The subsequent pacia will add the *correct PAC for* P, then flip bit p of the PAC to indicate that the input pointer was invalid (❸). $\mathcal{A}$ can now flip bit p back (❺) in order to obtain the correct PAC for pointer P (❻).

The PA signing gadget requires finding a matching ⟨autib, pacib⟩ pair operating on pointer P in the code without any use of P between these instructions. In PACStack each verification is immediately followed by a return, which ensures that the failure is detected. Tail-calls are a notable exception. Tail-calls are function calls executed before return and optimized so that the

[10] https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

```
...                  ; 𝒜 injects P at <ptr>             ❶
ldr     Xd, <ptr>    ; Xd ← P
autia   Xd, <mod>    ; Xd ← P*                          ❷
pacia   Xd, <mod>    ; Xd ← pacia (P, < mod >) ⊕ p      ❸
str     Xd, <ptr>    ; <ptr> ← Xd
...                  ; 𝒜 sets <ptr> to <ptr> ⊕ p        ❺
ldr     Xd, <ptr>    ; Xd ← pacia (P, < mod >)
autia   Xd, <mod>    ; Xd ← P (valid pointer)           ❻
```

Listing 9: PA adds a PAC based on the address bits. An invalid input pointer (❶), causes only a single bit-flip in the output PAC (❸). This could be exploited to generate valid PACs for arbitrary pointers.

```
A:
  epilogue:
    ...
    ldr    X28, [SP]   ; load invalid aret'_{i-1}
    autib  LR,  X28    ; LR ← ret*_i                    ③
    b      <B>         ; tail call B                     ①

B:
  prologue:
    str    X28, [SP]
    pacib  LR,  X28    ; LR ← aret_i ⊕ p                 ⑤
    ...
  epilogue:
    ...
    autib  LR,  X28    ; LR ← ret*_i                    ④
    ret                ;                                 ②
```

Listing 10: Tail-call optimizations on 64-bit ARM remove an unnecessary return by converting a branch with link instruction to a non:vsp -linking branch instruction (①).

callee directly returns to the caller of the optimized invocation of B in Listing 10. On 64-bit ARM processors, tail-calls are implemented using the b or br instructions that do not update LR (①). The tail-called function can return (②) to the LR value set before the tail-call (③). PACStack limits $\mathcal{A}$ to modifying the previous *auth* token on the stack. $\mathcal{A}$ could attempt to exploit the signing gadget to trick PACStack to accept an invalid $aret'_{i-1}$ (④), and subsequently load it into LR after return. This is not possible as $\mathcal{A}$ cannot flip the bit p of $aret'_i$ (⑤), because $aret_i \oplus p$ is: 1) kept in LR while in B, and 2) verified against $aret_{i+1}$ on subsequent function calls from B. The invalid $aret'_{i-1}$ is thus rejected by autib (④) before the return from B.

7.2.2 Sigreturn-oriented programming. Sigreturn-oriented programming [9] is a exploitation technique in UNIX-like operating systems, including Linux, that abuses the *signal frame* to take complete control of a process's execution state, i.e., the values of general purpose registers, SP, program counter (PC), status flags, etc. When the kernel delivers a signal, it suspends the process and changes the user-space processor context such that the appropriate signal handler is executed with the right arguments. When the signal handler returns, the original user-space processor context is restored. In a sigreturn attack $\mathcal{A}$ sets up a fake signal frame and initiates a return from a signal that the kernel never delivered. Specifically, a program returns from the handler using a sigreturn system call that reads a signal frame (struct sigcontext in Linux) from the process stack. A sigreturn attack is problematic for PACStack, as if successful, it would allow $\mathcal{A}$ control of any EL0 register, including CR.

A number of defense strategies against sigreturn attacks have been proposed for the Linux kernel. Bosman and Bos [9] propose placing keyed signal canaries in the signal frame that are validated by the kernel before performing a sigreturn, or to keep a counter of the number of currently executing signal handlers. However, modern Linux versions rely solely on address space layout randomization (ASLR) [33] to make it difficult for the attacker to trigger an unwarranted sigreturn. Fortunately sigreturn is never called directly from program code (in fact the GNU C library sigreturn simply returns an error value). Instead the system call is triggered by signal trampoline code placed either in the kernel's virtual dynamic shared object (vdso) or in the C library, both subject to ASLR. For our chosen adversary model (Section 3) ASLR is not sufficient as $\mathcal{A}$ can determine the contents of any readable memory in the process memory space. However, PACStack itself, together with coarse-grained CFI (Assumption **A2**), ensures that $\mathcal{A}$ cannot divert control flow from program code to the signal trampoline. Nonetheless, 64-bit ARM programs that might call system calls directly using the svc instruction (without going through C library system call wrappers), would not be protected against the presence of such gadgets. We discuss a potential general solution against sigreturn attacks that utilizes the ACS construction in Appendix B.

8 PERFORMANCE EVALUATION

At present, the only publicly available PA-enabled SoCs are the Apple A12 and S4, neither of which support PA for 3rd party code at the time of writing. To verify the correctness of instrumentation we ran all benchmarks on the ARMv8-A *Base Platform Fixed Virtual Platform (FVP)*, based on Fast Models 11.4, which supports ARMv8.3-A [4]. Because the FVP runs the v4.14 kernel, we have used PA RFC patches[11] modified to support all PA keys.

The FVP is not cycle-accurate and executes all instructions in one master cycle; therefore, it cannot be used for performance evaluation. Based on prior evaluations of the QARMA cipher [7], which is used as the underlying cryptographic primitive in reference implementations of PA [47], Liljestrand et al. estimate that the PAC calculations incur an average overhead of four cycles on a 1.2GHz CPU [35]. We employ the *PA-analog* (Listing 11)introduced by Liljestrand et al. to estimate the run-time overhead of PACStack. The PA-analog consists of four eor instructions that both read and write the registers used by the corresponding PA instruction in order to induce similar constraints on instruction pipelining within the CPU. To preserve compiler behavior, the PA-analog is swapped-in during a separate pre-emit pass, i.e., after both register allocation and instruction scheduling.

Using the PA-analog, we conducted benchmarks on a 96board Kirin 620 HiKey (LeMaker version) with an ARMv8-A Cortex A53 Octa-core CPU (1.2GHz) / 2GB LPDDR3 SDRAM (800MHz) / 8GB eMMC, running the Linux kernel v4.18.0 and BusyBox v1.29.2.

[11] https://lwn.net/Articles/752116/

```
; replace:
;  pacia   LR, CR
; with:
  eor      LR, LR, #const1
  eor      LR, LR, #const2
  eor      LR, LR, #const3
  eor      LR, LR, CR
```

Listing 11: PA-analog used to simulate overhead on non-PA hardware, based on an estimated overhead of 4 cycles. Three exclusive-or inputs are constants, whereas the last instruction uses both inputs to ensure instruction pipelining must get both values.

We have performed benchmarks using both nbench-byte-2.2.3[12] program and the SPEC CPU 2017 benchmark package[13].

8.1 nbench-byte-2.2.3

The nbench program includes 10 separate benchmarks and is designed to measure CPU and memory performance. The benchmarks employ dynamic workload adjustment to ensure that a test run takes at least a certain amount of time. In order to determine the relative overhead introduced by PACStack, we took the same approach as prior work [10, 35] and modified nbench to perform a pre-determined number of iterations of each benchmark and measured the execution time of each separately. All binaries used in the performance evaluation were produced by our PACStack-enabled compiler. We disabled all optimizations when compiling benchmark binaries (`-O0` flag for Clang and LLVM, and `-O=0` for `llc`). We evaluated the performance of nbench in three configurations: i) PACStack disabled, to determine the baseline execution time; ii) PACStack enabled, without PAC masking; and iii) PACStack enabled, with PAC masking. We repeated each benchmark 10 times and measured the user time using the `time` utility for each benchmark run. The results are shown in Figure 6, and indicate an overhead of 0.5% when using PAC masking, and an overhead of $< 0.3\%$ without (geometric mean of all benchmarks [52]).

8.2 SPEC CPU 2017

In contrast to nbench-byte-2.2.3, SPEC CPU 2017 is an industry-standard benchmarking suite that consists of larger units of work based on real-world applications. Due to resource constraints it was not feasible to install both the PACStack compiler and the SPEC CPU suite on the FVP or HiKey board. Instead we compiled the benchmarks with the SPEC `runcpu` utility configured to use WLLVM[14] as the compiler. WLLVM produces binaries containing the LLVM Intermediate Representation (IR), which we extracted and instrumented using PACStack and the PA-analog.

For comparison, we also measured the run-time overhead of *ShadowCallStack* [15], an instrumentation pass added in LLVM/-Clang 7.0. ShadowCallStack protects programs against return address overwrites by saving a function's return address in the function prolog to a separately allocated shadow stack and checking the return address on the stack against the shadow stack in the function epilog. On 64-bit ARM the instrumentation makes use of `X18` register to reference the shadow stack. Currently the runtime support for the ShadowCallStack instrumentation is only available in Android's Bionic C library. In addition, ShadowCallStack is only compatible with uninstrumented libraries which reserve the `X18` register, i.e., binaries built for a platform whose ABI reserves `x18`, (e.g., Android, Darwin, Fuchsia and Windows) or are compiled with the `-ffixed-x18` flag. To be able to perform a fair comparison against PACStack instrumented SPEC using the GNU C library (`glibc`) we ported ShadowCallStack support to `glibc`version 2.23 and compiled versions of our modified `glibc` and `libgcc` 6.4.1, the GCC low-level runtime library with the `-ffixed-x18` flag. Our changes to `glibc` were based on revision da772e2[15] of the Bionic C library. For PACStack measurements we used a prebuilt version of `glibc` 2.23 and `libgcc` 6.4.1 distributed by Linaro[16].

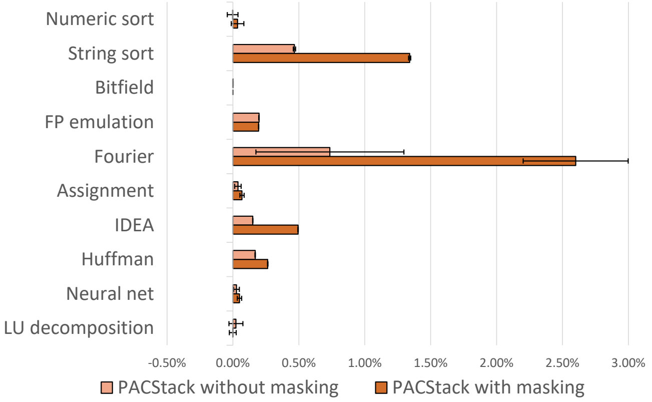


Figure 6: Relative performance overhead of the individual nbench-byte-2.2.3 benchmarks. The error bars indicate the standard error for $n = 10$ test runs per benchmark. The geometric mean of all benchmarks is 0.5%.

All benchmarks were compiled with the `-O0` flag to disable optimizations. The benchmark execution command and input files were determined using the SPEC `specinvoke` utility and then timed on the HiKey board using `time`.

Our measurements include all C-language SPECrate benchmarks, with the exception of two benchmarks that were incompatible with the WLLVM build environment that we used. For each benchmark, we compared the performance of the baseline (with PACStack and ShadowCallStack disabled) with three different configurations: i) PACStack without masking, ii) PACStack with masking, and iii) ShadowCallStack. Results are shown in Figure 7 and are reported as the mean overhead (w.r.t the baseline) and corresponding standard error. The SPEC CPU 2017 benchmark suite is resource intensive [42]; a single iteration of all SPEC benchmarks in Figure 7 took 13 times longer than an iteration of all nbench benchmarks. We therefore performed fewer measurements for SPEC than for nbench. Consequently, though the SPEC benchmarks are more representative of real-world workloads, they are more sensitive to

[12] http://www.math.utah.edu/~mayer/linux/bmark.html
[13] https://www.spec.org/cpu2017
[14] https://github.com/travitch/whole-program-llvm
[15] https://android.googlesource.com/platform/bionic/+/da772e2113fad40575eea4ebbb522509be7dfe4f%5E%21/
[16] https://releases.linaro.org/components/toolchain/binaries/6.4-2018.05/

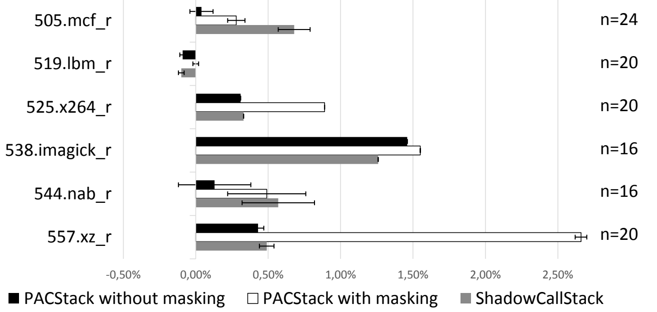


Figure 7: Relative performance overhead for SPEC CPU 2017 benchmarks; error bars show the standard error for n measurements. $n = 20$ for the ShadowCallStack configuration.

outliers than those in Figure 6. The results show PACStack incurs an overhead of 0.9% with masking, and 0.4% without masking (geometric mean of all benchmarks). PACStack without masking performs marginally better than ShadowCallStack, which incurs an overhead of 0.5% (geometric mean of all benchmarks). The performance overhead of PACStack is proportional to the frequency of function calls; benchmarks with few function calls are affected less by the instrumentation compared to benchmarks with frequent function calls. For instance, the `519.lbm_r` benchmark performs fluid dynamics and consists of large nested loops with few function calls. Consequently we see little effect on performance in `519.lbm_r`; in fact, our measurements show a small improvement in performance, which is likely caused by CPU pipeline optimizations that happen to be advantageous. We observe the same behavior for ShadowCallStack.

Based on these results, we expect the overhead for both PACStack configurations to be a) comparable to ShadowCallStack, and b) negligible on ARMv8.3-A PA-capable hardware.

9 RELATED WORK

Control-flow hijacking attacks were discovered and popularized more than two decades ago [49]. The majority of CFI solutions proposed since then are *stateless*: they validate each control-flow transfer in isolation without distinguishing among different paths in the control-flow graph (CFG). *Fully-precise static CFI* [12] is in theory the most restrictive stateless policy that is possible without breaking the intended functionality of the protected program. In fully-precise static CFI, and by extension any stateless policy,the best possible policy for return instructions is to allow returns within a function F to target any instruction that follows a call to F. All stateless CFI schemes, including fully-precise static CFI, are vulnerable to *control-flow bending* [12].

Stateful CFI can express policies that take previous control-flow transfers into account. HAFIX [19] is a hardware-assisted CFI scheme that confines function returns to active call sites. Context-sensitive CFI [20, 27, 53] further ensures that *each* control-flow transfer taken by the program is consistent with a non-malicious trace. This leads to a more precise policy compared to stateless CFI, but context-sensitive CFI enforcement has been dismissed as impractical for real-world adoption [1]. Hardware-assisted branch recording features available in modern 64-bit Intel microprocessors show promise in enabling context-sensitive CFI enforcement on commodity hardware, but suffer from i) limited branch history used to make CFI decisions, ii) over-approximation of the program CFG, iii) reliance on complex run-time monitoring. HAFIX, on the other hand, requires changes to the underlying processor architecture.

Stateless forward-edge CFI enforcement is often combined with a *shadow stack* [1, 14–18, 22, 23, 28, 39, 40, 51] to enforce the integrity of return addresses stored on the call stack. In fact, the results by Carlini et al. [12] show that a shadow stack (or equivalent mechanism) is essential for the security of CFI. The shadow stack maintains a copy of each return address in a separate region of memory. Each return instruction is then instrumented to validate that the return addresses on the call and shadow stack match. This ensures that each return is restricted only to its corresponding call site.

Although shadow stacks provide precise protection, traditional shadow stacks incur significant performance overhead and lead to false positives for programming constructs that cause mismatches between calls and returns (C++ exceptions with stack unwinding, `setjmp`/ `longjmp`). Recent shadow designs demonstrate that performance can be increased by either leveraging a parallel shadow stack [17], or using a dedicated register for shadow stack addressing [11]. However, in these schemes the shadow stack still resides in the same address space as the target application, and can be compromised if the shadow stack location is known to $\mathcal{A}$. For traditional shadow stacks, a typical solution for dealing with mismatches between calls and returns is to pop return addresses off the shadow stack until a match is found, or the shadow stack is empty (e.g., binary RAD [14]). This not only increases the complexity and run-time of the shadow stack instrumentation placed in the function epilogue, but also sacrifices precision, e.g., it allows $\mathcal{A}$ to redirect `longjmp` to any previously active call site. This can be avoided by storing and validating both the return address and stack pointer [16, 41, 51]. So far, only hardware-assisted shadow stacks promise to achieve negligible overhead without security trade-offs (e.g., Intel CET[28]).

Park et al. [44] present a microarchitectural shadow stack implemenation that utilizes the branch predictor *return address stack*, a common hardware feature found in modern speculative superscalar processor designs. The return address stack is typically a circular buffer, so to avoid the loss of stored return addresses when the maximum capacity is reached, Park et al. modify the return address stack to spill a portion of it's content to backup storage in main memory. They use a Merkle tree caching scheme to efficiently authenticate the backup storage before it is read back to the return address stack. The latency of spill / fill operations on backup memory is effectively offset by the 100% hit rate for branch prediction thanks to the ability to retain return addresses that exceed the return address stack capacity.

The idea of using of MACs to protect the return address at run-time was introduced in *Cryptographic CFI* (CCFI) [37] which uses MACs to protect return addresses and other control-flow data (e.g., function pointers and C++ vtable pointers). CCFI's return address protection is similar to PA-based return address signing [47]; both bind the return address to the address of the function's stack frame and thus provide only coarse-grained resistance against pointer reuse attacks [35].

Program Counter Encoding [34, 43, 46] protects return addresses on the stack by encoding them with either a register-resident secret key [34], the SP [46], or the address at which the return address itself is stored (a.k.a. the *self-address*) [43]. It's efficient, but relying on a userspace-resident secret key makes such encoding schemes susceptible to buffer over-reads, and SP or self-address encoding suffer the same drawbacks as -msign-return-address [35, 47] (Section 2.2.1).

Other prominent defenses against control-flow attacks include fine-grained code randomization [33], and code-pointer integrity (CPI) [31]. Code randomization makes it more difficult for $\mathcal{A}$ to find suitable gadgets to exploit in their attacks, but is not effective if the memory layout of the program becomes known. CPI protects code pointers by storing them in a separate *safe stack*. The safe stack requires similar integrity guarantees as shadows stacks to remain effective [21].

PACStack targets the ARM architecture, which traditionally has received less attention compared to the x86 family of computer architectures in terms of CFI research. *MoCFI* [18] is a software-based CFI approach specifically targeting ARM application processors used in smartphones. It uses a combination of a shadow stack, static analysis and run-time heuristics to determine the set of valid targets for control-flow transfers, but suffers from the same drawbacks that plague traditional shadow stack schemes. *CFI CaRE* [40] is a CFI solution targeting small, embedded ARM-based microcontrollers (MCUs). It uses the ability to perform hardware-enforced isolated execution on ARMv8-M MCUs to isolate the shadow stack to a secure processor state. The ARMv8-M [5] architecture enforces that calls to secure functions must target *secure gate instructions* placed at the beginning of such functions. The ARMv8.5-A architecture introduces similar *branch target indicators* (BTI) [3] to also ARM application processors. BTI constitutes one way to meet the PACStack pre-requisite of coarse-grained CFI for indirect branch instructions, e.g., calls via function pointers.

10 DISCUSSION

10.1 Generalizing ACS to other data structures

ACS builds on the idea of chaining cryptographic authentication codes. This simple, yet powerful, construct is similar to hash chains, which have been used before as means of password protection (Lamport signatures [32]), digital signatures (Merkle trees [38]), and have seen use in technologies such as blockchain [54] and trusted hardware access control authorization policies [6].

While the focus of this work is on applying this idea to protect the integrity of return addresses in the program call stack, the same approach can be generalized to other data structures and applications. For example, the call-stack protection could easily be extended to cover the *frame pointer*, or other data stored in a function's stack frame, and protect such data from unauthorized modification.

In addition to instrumentation that can protect the call stack, an ACS-like authenticated stack, or other data structure such as a Merkle-tree [38] can be implemented as reusable library, which would allow application developers to protect the integrity of critical data structures from manipulation as a result of software [13, 26], or hardware attacks [29].

An example of such a use case is data structures in operating system kernels. For instance, the Linux kernel source code features a generic double linked list implementation, which doubles as a queue and stack, depending on where in the kernel it is used[17]. Kernel data structures are critical to the system security. Many of the vulnerabilities found in the kernel allow limited access to kernel data. Malicious modification of kernel data can lead to a wide range of effects, including privilege escalation and process hiding [8]. Applying ACS-like protection to critical kernel stacks can protect such structures from: i) malicious modification by $\mathcal{A}$ in an effort to compromise kernel data integrity ii) accidental misuse by programmers, e.g., operating on a stack as a queue and vice versa (a side-effect of reuse of generic list implementations).

10.2 Support for software exceptions

The setjmp / longjmp interface has traditionally been used to provide exception-like functionality in C. However, modern coding standards for C and C++ that aim to facilitate code safety, security, and reliability consider them harmful and forbid their use, e.g., MISRA C:2004 [25, Rule 20.7] and JSF AV C++ [36, Rule 20]. Recall from Section 5.5, that calling longjmp with an expired jmp_buf is is undefined behavior. For PACStack, this means that although the $aret_b$ in jmp_buf to the corresponding SP and $auth_i$, it cannot guarantee their *freshness*. $\mathcal{A}$ can modify jmp_buf to contain the previously used $aret_b$ and SP_b, but must also modify the stack-frame at SP_b, such that it contains the prior $aret_i$. This allows a control-flow transfer to a previously valid setjmp return site and SP value. To prevent reuse of expired jmp_buf buffers, longjmp can be rewound step-by-step, i.e., conceptually performing returns until the correct stack-frame is reached.

We plan to extend PACStack support to LLVM libunwind[18]libunwind performs frame-by-frame unwinding of the call stack. By validating the ACS on each stack frame unwinding, PACStack can ensure that a fresh and valid state is reached.

Because C++ exceptions also cause irregular stack unwinding they pose a similar challenge. However, C++ already performs more fine-grained stack unwinding to correctly destroy objects in unwound stack frames. The LLVM libcxxabi library will, depending on configuration, use libunwind for this purpose. With PACStack support in libunwind, we will be able to secure both setjmp / longjmp and support C++ exception handling.

10.3 Interoperability with unprotected code

Interoperability with unprotected (uninstrumented) code is an important deployment consideration. On one hand, a PACStack-protected application may need to interoperate with unprotected shared libraries. On the other, an unprotected app may need to interoperate with PACStack-protected shared libraries. The latter scenario is relevant for deployment in mobile operating systems such as Android, where multiple stakeholders provide application binaries to consumer devices. The deployment of PACStack, or any other run-time protection mechanism, is likely to be driven

[17] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/list.h?h=v5.0

[18] https://github.com/llvm/llvm-project/tree/master/libunwind

by OEMs that enable specific protection schemes for the operating system and system applications. However, OEMs are not in control of native code deployed as part of applications distributed through standard application marketplaces. It should be possible for one version of the shared libraries shipped with the operating system to remain interoperable with both PACStack-protected, and unprotected apps.

In Section 6.1 we explain how the use of callee-saved registers allows PACStack to remain interoperable with unprotected code. Recall that because `CR` is a callee-saved register it will be restored upon return. However, PACStack cannot guarantee that `CR` remains unmodified during the execution of the unprotected code that could temporarily store its value on the stack. To achieve the security guarantees describes in Section 7, PACStack instrumentation must be applied to both the application and any shared libraries. However, partial protection, e.g. PACStack-protected shared libraries can significantly raise the bar for the attacker, as calls into protected functions can still benefit from return address authentication. Common shared libraries like `libc` are a popular source for gadgets for run-time attacks because of their size and availability. Because functions in a PACStack-protected library validate the return address in returns from library functions, they effectively remove a potentially large set of reusable gadgets from $\mathcal{A}$'s disposal.

11 CONCLUSION

We showed how a general-purpose hardware security mechanism (ARM PA) can provide guarantees on-par with hardware-assisted shadow stacks, *without requiring additional hardware support or compromising security*. Other general-purpose primitives like memory tagging and branch target indicators are being rolled out. Creative uses of such primitives hold the promise of significantly improving software protection.

ACKNOWLEDGMENTS

This work was supported in part by the Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] Martín Abadi et al. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (Nov. 2009), 4:1–4:40.

[2] ARM Ltd. 2017. ARMv8 Architecture Reference Manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[3] ARM Ltd. 2018. Arm A-Profile Architecture Developments 2018: Armv8.5-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[4] ARM Ltd. 2018. Fast Models, Version 11.4, Fixed Virtual Platforms (FVP) Reference Guide. https://static.docs.arm.com/100966/1104/fast_models_fvp_rg_100966_1104_00_en.pdf.

[5] ARM Ltd. 2019. Armv8-M Architecture Reference Manual (ARM DDI 0553B.g). https://static.docs.arm.com/ddi0553/bg/DDI0553B_g_armv8m_arm.pdf.

[6] Will Arthur and David Challener. 2015. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security* (1st ed.). Apress, Berkely, CA, USA.

[7] Roberto Avanzi. 2017. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Trans. Symmetric Cryptol.* 2017, 1 (2017), 4–44.

[8] Ahmed M. Azab et al. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proc. ACM CCS '14*. 90–102.

[9] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *Proc. IEEE S&P '14*. 243–258.

[10] Ferdinand Brasser et al. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. https://arxiv.org/abs/1709.09917.

[11] Nathan Burow et al. 2019. SoK: Shining Light on Shadow Stacks. `arXiv:1811.03165v2 [cs.CR]`. https://arxiv.org/abs/1811.03165v2

[12] Nicolas Carlini et al. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proc. USENIX Security '15*. 161–176.

[13] Shuo Chen et al. 2005. Non-control-data Attacks Are Realistic Threats. In *Proc. USENIX Security '05*. 177–191.

[14] Tzi-Cker Chiueh and Fu-Hau Hsu. 2001. RAD: a compile-time solution to buffer overflow attacks. In *Proc. 21st International Conference on Distributed Computing Systems*. 409–417.

[15] Clang 7.0 Documentation. 2018. ShadowCallStack. https://releases.llvm.org/7.0.0/tools/clang/docs/ShadowCallStack.html.

[16] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. 2005. Using DISE to Protect Return Addresses from Attack. *ARM SIGARCH Comput. Archit. News* 33, 1 (2005), 65–72.

[17] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proc.ACM ASIA CCS '15*. 555–566.

[18] Lucas Davi et al. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc. NDSS '12*.

[19] Lucas Davi et al. 2015. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proc. ACM/EDAC/IEEE DAC '15*. 74:1–74:6.

[20] Ren Ding et al. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proc. USENIX Security '17*. 131–148.

[21] I. Evans et al. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proc. IEEE S&P '15*. 781–796.

[22] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. 2002. Detecting Manipulated Remote Call Streams. In *Proc. USENIX Security '02*. 61–79.

[23] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. 2004. Efficient context-sensitive intrusion detection. In *Proc. NDSS '04*.

[24] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. 2016. Dynamic Canary Randomization for Improved Software Security. In *Proc. ACM CISRC '16*. 9:1–9:7.

[25] HORIBA MIRA Ltd. 2004. Guidelines for the Use of the C Language in Critical Systems. http://www.misra.org.uk/

[26] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proc. IEEE S&P '16*. 969–986.

[27] Hong Hu et al. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proc. ACM CCS '15*. 1470–1486.

[28] Intel. 2016. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[29] Yoongu Kim et al. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proc. IEEE ISCA '14*. 361–372.

[30] Tim Kornau. 2009. *Return Oriented Programming for the ARM Architecture*. Ph.D. Dissertation. Ruhr-Universität Bochum.

[31] Volodymyr Kuznetsov et al. 2014. Code-pointer Integrity. In *Proc. USENIX OSDI '14*. 147–163.

[32] Leslie Lamport. 1981. Password Authentication with Insecure Communication. *Commun. ACM* 24, 11 (1981), 770–772.

[33] Per Larsen et al. 2014. SoK: Automated Software Diversity. In *Proc. IEEE S&P '14*. 276–291.

[34] Gyungho Lee and Akhilesh Tyagi. 2000. Encoded Program Counter: Self-Protection from Buffer Overflow Attacks.. In *Proc. CSREA ICIC '00*. 387–394.

[35] Hans Liljestrand et al. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proc. USENIX Security '19*.

[36] Lockheed Martin Corporation. 2005. Joint Strike Fighter Air Vehicle C++ Coding Standards (Revision C). http://www.jsf.mil/downloads/down_documentation.htm

[37] Ali Jose Mashtizadeh et al. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proc. ACM CCS '15*. 941–951.

[38] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*. Springer-Verlag, 369–378.

[39] Danny Nebenzahl, Mooly Sagiv, and Avishai Wool. 2006. Install-Time Vaccination of Windows Executables to Defend Against Stack Smashing Attacks. *IEEE Trans. Dependable Secur. Comput.* 3, 1 (2006), 78–90.

[40] Thomas Nyman et al. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *Research in Attacks, Intrusions, and Defenses*. 259–284.

[41] H. Ozdoganoglu et al. 2006. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285.

[42] R. Panda et al. 2018. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?. In *Proc. IEEE HPCA '18*. 271–282.

[43] Seho Park, Jongmin Lee, Yongsuk Lee, and Gyungho Lee. 2015. Control Flow Hardening with Program Counter Encoding for ARM® Processor Architecture. In *Proceedings of the International Conference on Communications and Computers (CC 2019) (Recent Advances in Electrical Engineering)*. 32–38.

[44] Yong-Joon Park and Gyungho Lee. 2004. Repairing Return Address Stack for Buffer Overflow Protection. In *Proc. '04*. 335–342.

[45] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2015. DynaGuard: Armoring Canary-based Protections Against Brute-force Attacks. In *Proc. ACM ACSAC '15*. 351–360.

[46] Changwoo Pyo and Gyungho Lee. 2002. Encoding Function Pointers and Memory Arrangement Checking Against Buffer Overflow Attack. In *Proc. ICICS '02*. 25–36.

[47] Qualcomm. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[48] Nigel P Smart. 2016. *Cryptography Made Simple*. Springer.

[49] Solar Designer. 1997. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[50] László Szekeres et al. 2013. SoK: Eternal War in Memory. In *Proc. IEEE S&P '13*. 48–62.

[51] Caroline Tice et al. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proc. USENIX Security '14*. 941–955.

[52] Erik van der Kouwe et al. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. https://arxiv.org/abs/1801.02381.

[53] Victor van der Veen et al. 2015. Practical Context-Sensitive CFI. In *Proc. ACM CCS '15*. 927–940.

[54] Dylan Yaga et al. 2018. *Blockchain Technology Overview*. Technical Report NIST.IR.8202. National Institute of Standards and Technology.

A SECURITY PROOFS

In Section 7.1, we gave an informal analysis of the security of ACS; here we give a more detailed proof of security, and in particular prove that authentication token masking prevents $\mathcal{A}$ from obtaining exploitable authentication token collisions.

The argument proceeds as follows: we suppose that $\mathcal{A}$, after obtaining q authentication tokens, can find a pair of inputs (x, y) and (x, y') whose authentication tokens $\mathsf{H}_\mathsf{K}(\cdot,\cdot)$ collide. This can be used to construct a distinguisher of the masks $\mathsf{H}_\mathsf{K}(0,\cdot)$ from a random string. The structure of the authentication tags is such that this further reduces to a semantic security game for one-time pad encryption of the masks. Then, we show that any violation of the integrity of an ACS-protected call stack also yields values whose authentication tokens collide as described above, allowing us to bound the probability of an integrity violation.

We summarize our notation in Table 2.

Theorem A.1 (PAC-masking prevents collision-finding). *Suppose that after q queries, an adversary $\mathcal{A}$ can distinguish $\mathsf{H}_\mathsf{K}(\cdot,\cdot)$ from a random oracle with advantage no greater than $Adv^{\mathcal{A}}_{PAC\text{-}Distinguish}(1^\lambda, H, q)$, as given in Figure 9. Then, assuming a key-length of λ for $\mathsf{H}_\mathsf{K}(\cdot,\cdot)$, and given access to q masked authentication tokens, $\mathcal{A}$ can identify a pair of inputs $(\hat{x}, \hat{y})$ and $(\hat{x}, \hat{y}')$ whose corresponding unmasked authentication tokens collide with advantage at most $2Adv^{\mathcal{A}}_{PAC\text{-}Distinguish}(1^\lambda, H, q)$.*

Proof. We begin with a collision-game $G^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$, shown in Figure 8 in which the adversary is given oracle access to the authentication token generator and then asked to provide values x, y, y' such that $\mathsf{H}_\mathsf{K}(x, y) = \mathsf{H}_\mathsf{K}(x, y')$.

An adversary that selects (x, y, y') at random from $\{0,1\}^{\mathsf{VA\_SIZE}} \times \{0,1\}^{\mathsf{VA\_SIZE}+b} \times \{0,1\}^{\mathsf{VA\_SIZE}+b}$, such that $y \neq y'$, will win with probability 2^{-b}; $\mathcal{A}$'s advantage is therefore

$$\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q) = \mathbb{P}\left[G^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q) = 1\right] - 2^{-b}.$$

Games	
G_{ACS} *(Figure 13)*	Security game for ACS integrity.
$G_{\text{PAC-Collision}}$ *(Figure 8)*	Security game for the identification of colliding authentication tokens.
$G_{\text{PAC-Distinguish}}$ *(Figure 9)*	Security game for the distinguishability of $\mathsf{H}_\mathsf{K}(\cdot,\cdot)$ from a random oracle.
G_1, G_2, G_3 *(Figure 11)*	Semantic security games for the mask $\mathsf{H}_\mathsf{K}(0,\cdot)$.

Adversary interfaces		
G_{ACS}	$\mathcal{A}_{\text{oracle-request}}$	Get path through the call-graph for which $\mathcal{A}$ wants the final authenticated return address pushed to the stack.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authenticated return address.
	$\mathcal{A}_{\text{ACS-Violation}}$	Return to the challenger authenticated return values that can be used to violate call stack integrity.
$G_{\text{PAC-Collision}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants a masked authentication token.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested masked authentication token.
	$\mathcal{A}_{\text{gen-collision}}$	Return to the challenger two authenticated return values with colliding authentication tokens.
$G_{\text{PAC-Distinguish}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants an authentication tag.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authentication token.
	$\mathcal{A}_{\text{distinguish}}$	Return to the challenger a single bit identifying whether the given tokens were from a random oracle or $\mathsf{H}_\mathsf{K}(\cdot,\cdot)$.
G_1, G_2	$\mathcal{B}_{\text{distinguish}}$	Identify the authentication token function used to generate masked authentication tokens.
G_3	$\mathcal{B}_{\text{distinguish'}}$	As for G_1, G_2, but with the inputs represented as strings rather than functions.

Table 2: Notation used in Appendix A.

```
$G^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$
$K \xleftarrow{\$} \{0,1\}^\lambda$
/ Give $\mathcal{A}$ $q$ masked authentication tokens
/ of their choice.
for $i \in \{1, \ldots, q\}$ do
    $(x, y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
    $\mathcal{A}_{\text{oracle-response}}\ (\mathsf{H_K}(x, y) \oplus \mathsf{H_K}(0, y))$
endfor
/ $\mathcal{A}$ is challenged to provide inputs whose authentication tokens collide.
$(\hat{x}, \hat{y}, \hat{y}') \leftarrow \mathcal{A}_{\text{gen-collision}}()$
if $\hat{y} \neq \hat{y}' \wedge \mathsf{H_K}(\hat{x}, \hat{y}) = \mathsf{H_K}(\hat{x}, \hat{y}')$ then
    return 1
else
    return 0
endif
```

Figure 8: Security game for finding colliding PACs given masked authentication tokens.

```
$G^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q)$
$K \xleftarrow{\$} \{0,1\}^\lambda$

/ $\mathcal{B}$ is given values of their choice from either
/ $\mathsf{H_K}(\cdot, \cdot)$ or a random oracle $RO(x, y)$
$S_0(x, y) \overset{def}{=} RO(x, y)$
$S_1(x, y) \overset{def}{=} \mathsf{H_K}(x, y)$
$c \xleftarrow{\$} \{0, 1\}$
for $i \in \{1, \ldots, q\}$ do
    $(x, y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
    $\mathcal{A}_{\text{oracle-response}}\ (S_c(x, y))$
endfor

/ $\mathcal{A}$ is challenged to determine whether it received
/ values from $\mathsf{H_K}(\cdot, \cdot)$ or the random oracle.
$\hat{c} \leftarrow \mathcal{A}_{\text{distinguish}}()$
if $c \neq \hat{c}$ then
    return 1
else
    return 0
endif
```

Figure 9: Security game in which $\mathcal{A}$ attempts to distinguish $\mathsf{H_K}(\cdot, \cdot)$ from a random oracle.

We will bound this advantage by reduction to a semantic security game for the masks. We consider the following games, shown in Figure 11, and described in Figure 10.

The first hop, from G_1 to G_2, is based on indistinguishability and relaxation: we suppose that $\mathsf{H_K}(\cdot, \cdot)$ can be distinguished from a random oracle with probability no more than $\frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q)$, and that the adversary is not limited in the number of queries that can be made to the masked authentication token oracle. Then,

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda, H, q) = 1] \leq \mathbb{P}[G_2^{\mathcal{A}}(1^\lambda, H, q) = 1] + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q).$$

$G_1^{\mathcal{B}}(1^\lambda, H, q)$**:** $\mathcal{B}$ obtains masked authentication tokens $\mathsf{H_K}(x, y) \oplus \mathsf{H_K}(0, y)$ for up to q pairs (x, y) of $\mathcal{B}$'s choice, and must then distinguish the masks $\mathsf{H_K}(0, \cdot)$ from a random oracle.

$\mathsf{H_K}(\cdot, \cdot) \rightarrow$ random oracle

$G_2^{\mathcal{B}}(1^\lambda, H, q)$**:** This is the same as the previous game, except that $\mathsf{H_K}(\cdot, \cdot)$ is replaced by a random oracle and $\mathcal{B}$ is not limited in their number of queries. $\mathcal{B}$ must now distinguish between two random oracles, one of which is used in computing the authentication tokens, and one of which is independent of the authentication tokens.

Reformulation

$G_3^{\mathcal{B}}(1^\lambda, H, q)$**:** This is the semantic security game for repeated one-time-pad encryptions of a random string.

Figure 10: The game-hops used in Figure 11.

The second hop, from G_2 to G_3, is a mere reformulation of G_2 such that random oracles are represented as strings, and that rather than allowing $\mathcal{B}$ to request arbitrarily many authentication tokens from the challenger, we instead give $\mathcal{B}$ direct access to the oracle, as represented by the sequence of strings $T_{1\ldots 2^{\mathsf{VA\_SIZE}}}$.

The third game is a semantic security game for the one-time pad, where $\mathcal{A}$ is given $2^{\mathsf{VA\_SIZE}}$ encryptions of S_1 and then asked to distinguish between S_1 and a random string. The perfect secrecy of the one-time pad means that $\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda) = 1] = \frac{1}{2}$ and so

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda) = 1] \leq \frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q). \quad (2)$$

Finally, we provide a reduction from $\mathrm{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$ to $\mathrm{G}_1{}^{\mathcal{B}}(1^\lambda)$. Suppose $\mathcal{A}$ can win $\mathrm{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$ with advantage $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$. Then, we define an adversary $\mathcal{A}^{\mathcal{A}}$ for $\mathrm{G}_1{}^{\mathcal{B}}(1^\lambda)$, shown in Figure 12.

This adversary wins $\mathrm{G}_1{}^{\mathcal{B}}(1^\lambda)$ with probability at least $\frac{1}{2} + \frac{1}{2}\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$, and so by (2)

$$\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q) \leq 2\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q).$$

If the MAC $\mathsf{H_K}(\cdot, \cdot)$ is a pseudo-random function family with respect to K, then $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q)$ is negligible, and thus so is $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$. □

With a bound on $\mathcal{A}$'s probability of successfully obtaining a PAC collision, we may now obtain a bound on their probability of violating the integrity of an ACS-protected call stack.

$\mathsf{H}_\mathsf{K}(\cdot,\cdot) \longrightarrow$ random oracle

random oracle $\longrightarrow$ random string

```
$G_1^{\mathcal{B}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$
$S_0(y) \overset{def}{=} RO(y)$
$S_1(y) \overset{def}{=} \mathsf{H}_\mathsf{K}(0, y)$
$T(x, y), x \neq 0$, first $q$ queries $\overset{def}{=} \mathsf{H}_\mathsf{K}(x, y) \oplus \mathsf{H}_\mathsf{K}(0, y)$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T(\cdot,\cdot)$.
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
  return 1
else
  return 0
endif
```

```
$G_2^{\mathcal{B}}(1^\lambda, H, q)$

$S_0(y) \overset{def}{=} RO_0(y)$
$S_1(y) \overset{def}{=} RO_1(0, y)$
$T(x, y), x \neq 0 \overset{def}{=} RO_1(x, y) \oplus RO_1(0, y)$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T(\cdot,\cdot)$.
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
  return 1
else
  return 0
endif
```

```
$G_3^{\mathcal{B}}(1^\lambda, H, q)$

$P_{1\ldots 2^{\text{VA\_SIZE}}} \leftarrow \{0,\ldots,2^b-1\}^{2^{b+\text{VA\_SIZE}}}$
$S_0 \xleftarrow{\$} \{0,\ldots,2^b-1\}^{2^{b+\text{VA\_SIZE}}}$
$S_1 \xleftarrow{\$} \{0,\ldots,2^b-1\}^{2^{b+\text{VA\_SIZE}}}$
$T_{1\ldots 2^{\text{VA\_SIZE}}} \leftarrow P_{1\ldots 2^{\text{VA\_SIZE}}} \oplus S_1$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T$....
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish'}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
  return 1
else
  return 0
endif
```

Figure 11: Security games used in Theorem A.1.

```
$\mathcal{B}^{\mathcal{A}}_{\text{oracle-request}}()$
return $\mathcal{A}_{\text{oracle-request}}()$

$\mathcal{B}^{\mathcal{A}}_{\text{oracle-response}}(x)$
$\mathcal{A}_{\text{oracle-response}}(x)$

$\mathcal{B}^{\mathcal{A}}_{\text{distinguish}}(T, S, S')$
$x, y, y' \leftarrow \mathcal{A}_{\text{gen-collision}}(T)$
if $S(y) \oplus S(y') = T(x, y) \oplus T(x, y')$ then
  return 1
else
  return 0
endif
```

Figure 12: An adversary $\mathcal{B}^{\mathcal{A}}$ for G_1 used in our black-box reduction of $G_{\textbf{PAC-Collision}}$ to G_1. Not shown is the variant $\mathcal{B}^{\mathcal{A}}_{\textbf{distinguish'}}(T, S, S')$ that is identical to $\mathcal{B}^{\mathcal{A}}_{\textbf{distinguish}}(T, S, S')$ except that T, S, and S' are given in the form of strings.

Theorem A.2 (Security of ACS). *Consider a program whose call stack is protected by ACS, which has a call-graph C and b-bit masked authentication tokens* $\mathsf{T}_K(x, y) = \mathsf{H}_K(x, y) \oplus \mathsf{H}_K(0, y)$. *Then, an adversary with arbitrary control over memory can violate backward-edge control-flow integrity with probability*

$$\mathbb{P}\left[G^{\mathcal{A}}_{ACS}(1^\lambda, H, C, q)\right] \leq \mathbb{P}\left[G^{\mathcal{A}}_{PAC\text{-}Collision}(1^\lambda, H, q)\right] \leq 2^{-b} + 2Adv^{\mathcal{A}}_{PAC\text{-}Distinguish}(1^\lambda, H, q)$$

Proof. We begin with a security game for ACS, shown in Figure 13.

Our goal is to provide a black-box reduction from $G^{\mathcal{A}}_{\text{ACS}}(1^\lambda, H, C, q)$ to $G^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$.

From line 24 of Figure 13, winning $G^{\mathcal{A}}_{\text{ACS}}$ implies that $\mathcal{A}$ has obtained colliding authentication tokens, and therefore $\mathcal{A}$ can win $G^{\mathcal{A}}_{\text{PAC-Collision}}$ with probability at least $\mathbb{P}[G^{\mathcal{A}}_{\text{ACS}}]$. Substituting the bound from Theorem A.1, we obtain the bound given. □

B MITIGATION OF SIGRETURN ATTACKS

A solution for precluding sigreturn attacks against PACStack would be to include the *signal return value* to the PACStack chain via the PC value stored on the signal frame:

$$asigret_i = \begin{cases} \texttt{pacib}(\texttt{PC} = sigret_i, asigret_{i-1}) & \text{if } i > 0 \\ \texttt{pacib}(\texttt{PC} = sigret_i, \texttt{CR} = aret_n) & \text{if } i = 0 \end{cases}$$

Upon signal delivery, the kernel stores a copy of $asigret_n$ securely in kernel space as a reference value. If the process was already executing a signal handler, and thus the kernel already has a reference copy of $asigret_{n-1}$ on record, it stores $asigret_{n-1}$ in the new signal frame and overwrites the secure copy with $asigret_n$. On `sigreturn` the kernel attempts to validate the PC and CR values in the signal frame as though the reference value was $asigret_0$. If successful it performs the signal return to $sigret_n$ and restores $aret_n$ to CR. Otherwise the kernel assumes a return to a nested signal handler, and retrieves $sigret'_n$ and $asigret'_{n-1}$ from the signal frame, validates them by calculating $asigret'_n = \texttt{pacib}(sigret'_n, asigret'_{n-1})$ and comparing the result against the stored $asigret_n$ reference value. If successful the kernel replaces $asigret_n$ with $asigret_{n-1}$ in the secure kernel store and performs the signal return to $sigret_n$. If the validation fails the kernel terminates the process. This prevents $\mathcal{A}$ from 1) overwriting CR, and 2) forging the PC values in signal frames. For general protection against sigreturn attacks corrupting any register stored in the signal frame, all register values could be included in the *asigret* calculation using the pacga instruction and validated at the time of `sigreturn`.

```
$G^{\mathcal{A}}_{\text{ACS}}(1^\lambda, H, C, q)$
 1 :  $K \xleftarrow{\$} \{0,1\}^\lambda$
 2 :
 3 :  / Give $\mathcal{A}$ $q$ tokens from call-graph traversals.
 4 :  for $i \in \{1, \ldots, q\}$ do
 5 :    $p_{1\ldots m+1} \leftarrow \mathcal{A}_{\text{oracle-request}}()$
 6 :    / Is the request for a real path through the call-graph?
 7 :    if $\exists j : p_j \rightarrow p_{j+1} \notin \text{edges}(C)$ then
 8 :      return 0
 9 :    endif
10 :    $auth_m \leftarrow \mathsf{T}_K(p_m, \mathsf{T}_K(p_{m-1}, \cdots) \parallel p_{m-1}) \parallel p_m$
11 :    $\mathcal{A}_{\text{oracle-response}}(auth_m)$
12 :  endfor
13 :
14 :  $ptr_{\text{jumper}}, ptr_{\text{correct}}, auth_{\text{correct}}, t_{\text{correct}},$
15 :      $ptr_{\text{adv}}, auth_{\text{adv}}, t_{\text{adv}} \leftarrow \mathcal{A}_{\text{ACS-Violation}}()$
16 :
17 :  / The substituted masked authenticated return address must be different.
18 :  if $ptr_{\text{correct}} = ptr_{\text{adv}} \wedge auth_{\text{correct}} = auth_{\text{adv}}$ then
19 :    return 0
20 :  endif
21 :
22 :  / Does the return pointer authenticate correctly with the adversary's
23 :  / new masked authenticated return address as the modifier?
24 :  if $\mathsf{H}_K(ptr_{\text{jumper}}, auth_{\text{correct}} \parallel ptr_{\text{correct}})$
25 :    $\neq \mathsf{H}_K(ptr_{\text{jumper}}, auth_{\text{adv}} \parallel ptr_{\text{adv}})$ then
26 :    return 0
27 :  endif
28 :
29 :  / Did the adversary provide a valid masked authenticated return address?
30 :  if $auth_{\text{adv}} = \mathsf{H}_K(ptr_{\text{adv}}, t_{\text{adv}})$
31 :    return 1
32 :  else
33 :    return 0
34 :  endif
```

Figure 13: Security game for ACS with respect to a program having call-graph C and authentication token function $\mathsf{T}_K(\cdot,\cdot)$.

C GLOSSARY

Return address	An address that is used as the target of a return instruction
Authenticated pointer	A pointer containing some embedded data that can be used to validate its authenticity
Authenticated return address	An authenticated pointer whose ‘pointer’ part is a return address.
PAC	The value $\mathsf{H}_\mathsf{K}(\cdot, \cdot)$ produced by ARM-PA.
Mask	The pseudo-random value $\mathsf{H}_\mathsf{K}(0, aret_{i-1})$
Authentication token	$\mathsf{H}_\mathsf{K}(ret_i, \mathsf{H}_\mathsf{K}(ret_{i-1}, \cdots) \parallel ret_{i-1})$ in ACS;$\mathsf{H}_\mathsf{K}(ret_i, \mathsf{SP})$ in [47]
Masked authentication token	The authentication token exclusive-OR-ed with the mask

D ARMv8.3-A PA INSTRUCTIONS

Instruction	Mnemonic	PA Key					Addr.	Mod.	Backwards-compatible
		Instr.		Data		Gen-eric			
		A	B	A	B				
BASIC POINTER AUTHENTICATION INSTRUCTIONS									
Add PAC to instr. addr.	`paciasp`	✓					LR	SP	✓
	`pacia`	✓					*Xd*	*Xm*	✗
	`paciaz`	✓					LR	*zero*	✓
	`paciza`	✓					*Xd*	*zero*	✗
	`pacia1716`	✓					X17	X16	✓
	`pacibsp`		✓				LR	SP	✓
	`pacib`		✓				*Xd*	*Xm*	✗
	`pacibz`		✓				LR	*zero*	✓
	`pacizb`		✓				*Xd*	*zero*	✗
	`pacib1716`		✓				X17	X16	✓
Add PAC to data addr.	`pacda`			✓			*Xd*	*Xm,*	✗
	`pacdza`			✓			*Xd*	*zero*	✗
	`pacdb`				✓		*Xd*	*Xm*	✗
	`pacdzb`				✓		*Xd*	*zero*	✗
Calculate generic MAC	`pacga`					✓			✗
Authenticate instr. addr.	`autiasp`	✓					LR	SP	✓
	`autia`	✓					*Xd*	*Xm*	✗
	`autiaz`	✓					LR	*zero*	✓
	`autiza`	✓					*Xd*	*zero*	✗
	`autia1716`	✓					X17	X16	✓
	`autibsp`		✓				LR	SP	✓
	`autib`		✓				*Xd*	*Xm*	✗
	`autibz`		✓				LR	*zero*	✓
	`autizb`		✓				*Xd*	*zero*	✗
	`autib1716`		✓				X17	X16	✓
Authenticate data addr.	`autda`			✓			*Xd*	*Xm*	✗
	`autdza`			✓			*Xd*	*zero*	✗
	`autdb`				✓		*Xd*	*Xm*	✗
	`autdzb`				✓		*Xd*	*zero*	✗
Strip PAC	`xpacd`						*Xd*		✗
	`xpaci`						*Xd*		✗
	`xpaclri`						LR		✓
COMBINED POINTER AUTHENTICATION INSTRUCTIONS									
Authenticate instr. addr. and return	`retaa`	✓					LR	SP	✗
	`retab`		✓				LR	SP	✗
Authenticate instr. addr. and branch	`braa`	✓					*Xd*	*Xm*	✗
	`braaz`	✓					*Xd*	*zero*	✗
	`brab`		✓				*Xd*	*Xm*	✗
	`brabz`		✓				*Xd*	*zero*	✗
Authenticate instr. addr. and branch with link	`blraa`	✓					*Xd*	*Xm*	✗
	`blraaz`	✓					*Xd*	*zero*	✗
	`blrab`		✓				*Xd*	*Xm*	✗
	`blrabz`		✓				*Xd*	*zero*	✗
Authenticate instr. addr. and exception return	`eretaa`	✓					ELR	SP	✗
	`eretab`		✓				ELR	SP	✗
Authenticate data. addr. and load register	`ldraa`				✓		*Xd*	*zero*	✗
	`ldrab`					✓	*Xd*	*zero*	✗

Table 3: List of PA instructions [35]. *PA Key* indicates the PA key the instruction uses. *Addr.* indicates the source of the address to be signed or authenticated. *Mod.* indicates the modifier used by the instruction. *Xd* and *Xm* indicates that the input is taken from a general purpose register. The *backwards-compatible* column indicates if the instruction is safe on pre ARMv8.3-A.