ISE SRA
Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz Universität Hannover

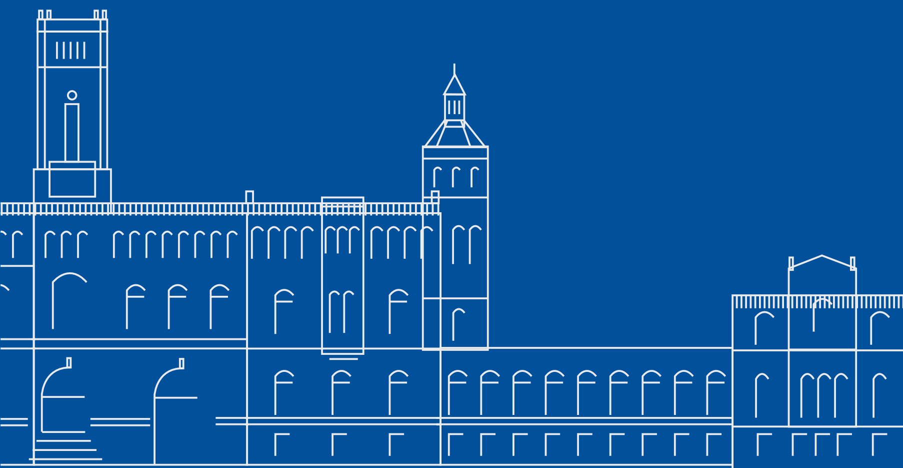
Illia Ostapyshyn

Software-Emulated Pointer Authentication for Control-Flow Integrity Protection

Bachelor's Thesis

September 23, 2022

Please cite as:
Illia Ostapyshyn, "Software-Emulated Pointer Authentication for Control-Flow Integrity Protection" Bachelor's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, November 2024.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Software-Emulated Pointer Authentication for Control-Flow Integrity Protection

Bachelorarbeit im Fach Elektro- und Informationstechnik

vorgelegt von

Illia Ostapyshyn

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Florian Rommel, M.Sc.**

Beginn der Arbeit:	**23. März 2022**
Abgabe der Arbeit:	**23. September 2022**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Illia Ostapyshyn)
Hannover, 23. September 2022

ABSTRACT

The shortcomings of existing Control-Flow Integrity techniques, in particular their significant performance overhead, have motivated the development of hardware-based solutions. One such protection technique is the pointer authentication code (PAC) mechanism introduced in the ARMv8.3-A architecture. Facilitated by hardware support, this feature provides efficient means for protection of code and data pointers with negligible performance and memory overhead. In line with the expectation that this technique will find wide adoption in the industry, mainstream operating systems and major compilers already support PAC. However, commercial off-the-shelf SoCs with hardware including this protection mechanism are currently sparse.

This work presents software-emulated pointer authentication mechanism implemented as a Linux kernel extension. To avoid the performance overhead of context switching, the communication between userspace applications and the OS extension does not involve system calls. Instead, the communication takes place exclusively via a shared memory area. The user fully commits one or more CPU cores to continuously poll the shared memory and process incoming pointer authentication requests. The kernel extension and the accompanying GCC plugin constitute a complete CFI solution for return address protection. To further mitigate the run-time overhead, the GCC plugin features several heuristics to omit the authentication for functions where a stack-buffer overflow is unlikely. The evaluation using the multithreaded memory-caching system *memcached* shows a moderate increase in average response latency of 2.7 % on x86-64 and 30.3 % on AArch64. This thesis builds upon and seeks to benefit related works related to the emulation of ARMv8.3-A pointer authentication by presenting solutions for support of multithreaded applications and improved code instrumentation plugin for return address protection.

KURZFASSUNG

Die Nachteile existierender Control-Flow-Integrity-Verfahren und insbesondere die erheblichen Leistungseinbußen, die von ihnen verursacht werden, haben die Entwicklung von hardwarebasierten Lösungen angestoßen. Der PAC-Mechanismus (Pointer Authentication Code) ist eine solche Schutztechnik, die als Teil der ARMv8.3-A Architektur eingeführt wurde. Durch Hardwareunterstützung bietet dieser Mechanismus ein effizientes Mittel, um Code- und Datenzeiger mit vernachlässigbaren Leistungs- und Speicher-Overhead zu sichern. Entsprechend der Erwartung, dass sich diese Technik in der Industrie durchsetzen wird, bieten etablierte Betriebssysteme und Compiler bereits Unterstützung für PAC. Allerdings sind kommerzielle und serienmäßig produzierte Ein-Chip-Systeme, die diesen Mechanismus implementieren, derzeit rar.

Im Rahmen dieser Arbeit wird eine durch Software emulierte Implementierung des PAC-Mechanismus als eine Erweiterung des Linux-Kerns präsentiert. Um den durch Kontextwechsel verursachten Leistungs-Overhead zu vermeiden, wird auf die Verwendung von Systemaufrufen bei der Kommunikation zwischen Userspace-Anwendung und Betriebssystem verzichtet. Stattdessen findet die Kommunikation ausschließlich über einen geteilten Speicherbereich statt. Der Benutzer widmet einen oder mehrere CPU-Kerne dem Zweck der kontinuierlichen Abfrage des geteilten Speicherbereichs und der folgenden Bearbeitung der Authentifizierungs-Anfragen. Die Kern-Erweiterung und das dazugehörige GCC-Plugin stellen eine vollständige Lösung für die Sicherung der Rücksprungadressen dar. Das GCC-Plugin bietet mehrere Heuristiken, um Funktionen, in denen ein Stapelüberlauf unwahrscheinlich ist, von der Authentifizierung auszunehmen und so den Laufzeit-Overhead weiter zu mindern. Die Auswertung wurde mittels des mehrfädigen Speicher-Caching-Systems *memcached* durchgeführt und zeigt eine mäßige Erhöhung der durchschnittlichen Latenz von 2.7 % bei x86-64- und 30.3 % bei AArch64-Systemen. Diese Arbeit baut auf vorherigen Arbeiten zur Emulation von PAC auf und versucht diese zu ergänzen, indem sie Lösungen für Unterstützung von mehrfädigen Anwendungen und ein verbessertes GCC-Plugin zum Schutz von Rücksprungadressen durch Code-Instrumentierung präsentiert.

CONTENTS

Abstract **v**

Kurzfassung **vii**

1 Introduction **1**

2 Fundamentals **3**
2.1 Virtual Memory Management 3
2.2 Shared-Memory Multiprocessing 4
2.2.1 Memory ordering 5
2.2.2 Atomicity 5
2.3 Software Vulnerabilities 6
2.3.1 Stack-based buffer overflow 6
2.3.2 Conventional mitigations 8
2.3.3 Control-Flow Integrity techniques 9
2.3.4 ARMv8.3-A pointer authentication 10
2.4 Code Generation in GCC 12
2.5 Related Work 13
2.5.1 Software-based protection of code pointers 13
2.5.2 Hardware-based techniques 13
2.5.3 Applications of pointer authentication 14
2.6 Summary 14

3 Architecture **17**
3.1 Communication Protocol 17
3.2 GCC Plugin 18
3.2.1 Return address protection on AArch64 19
3.2.2 Return address protection on x86-64 20
3.2.3 Configuration options 21
3.2.4 Compatibility with compiler optimizations 21
3.2.5 Code instrumentation 22
3.3 Linux Kernel Extension 24
3.3.1 Configuration options 24
3.3.2 Multiprocessing support 25
3.3.3 Kernel threads for pointer authentication 27
3.3.4 Modifications to the OS scheduler 28

3.4 Summary 29

4 Evaluation **31**
4.1 Protection Effectiveness 31
4.2 Synthetic Benchmarks 33
4.2.1 Impact of the hashing algorithms 33
4.2.2 Impact of the protection scopes 35
4.3 Multithreaded Application 37
4.3.1 Run-time performance and multicore scalability 37
4.3.2 Energy efficiency 40
4.4 Summary 41

5 Conclusion **43**

Lists **45**
List of Acronyms 45
List of Figures 47
List of Tables 49
List of Listings 51
Bibliography 53

1 INTRODUCTION

In pursuit of automation and following the rapid development of portable and affordable electronics, computers have found their way into many aspects of our lives. Nowadays, computerized systems perform tasks ranging from providing convenience services to management of safety-critical systems. While the latter applications require high reliability due to the involved risks, the former potentially handle confidential data. Therefore, computer software is expected to be resistant to attempts by malicious actors to alter the program's behavior or to bypass certain access restrictions. Real-world experience shows that computer programs are often susceptible to such attacks [1]. These attacks exploit *security vulnerabilities* — design flaws that the programmer left inadvertently during the development of the program. Programs written in low-level programming languages like C or C++ are especially prone to security vulnerabilities that can be used by the attacker to redirect the program's execution flow and gain unauthorized access to data or services running on the system.

The ever-growing complexity of computer software makes it increasingly hard to detect these vulnerabilities at the development stage [2]. The impracticality of this approach gave rise to protection techniques that aim to prevent exploitation of security vulnerabilities at runtime. These techniques, summarized under the term *Control-Flow Integrity* (CFI), are traditionally implemented in software and use code instrumentation to embed additional protections into the program. Unfortunately, the CFI techniques that provide strong security guarantees often come with a substantial performance overhead hindering their practical application [3].

High performance overhead associated with the present CFI approaches inspired the development of hardware-based techniques. A promising technique that was introduced in 2017 as part of the ARMv8.3-A architecture, is the *pointer authentication code* (PAC) mechanism [4]. As a result of its hardware-based implementation, this mechanism provides efficient means to protect control-flow elements by complementing them with a cryptographic signature. Although the mainstream compilers already provide support for PAC-based return address protection, it has not found wide use yet due to the limited availability of commercially produced ARM systems that implement the PAC mechanism.

This issue is addressed in the related PAC-PL project, where Serra et al. [5] emulated the PAC mechanism in hardware using a field-programmable gate array (FPGA) chip on heterogeneous platforms that do not support the original ARMv8.3-A pointer authentication natively. PAC-PL comes with a GNU Compiler Collection (GCC) plugin that applies this mechanism to protect return addresses on the stack. This thesis is motivated by concerns regarding the economic viability of the FPGA approach and explores the idea of implementing a similar mechanism purely in software and in tight integration with the OS. The central concept is to dedicate one or multiple CPU cores solely to the pointer authentication service and facilitate its communication with the

userspace applications using a shared memory area. Apart from implementing this concept in a Linux kernel extension, this work seeks to benefit the original PAC-PL project by addressing its missing support for multithreading and improving the original code instrumentation plugin for return address protection.

This thesis consists of five chapters. Following this introduction (Chapter 1), Chapter 2 introduces the concepts required to understand the architecture of the solution. These include fundamentals of virtual memory paging, memory access ordering, as well as foundations of low-level software security. Chapter 3 provides an in-depth information on the architecture of the pointer authentication extension for the Linux kernel and the design of the GCC plugin for return address protection. In Chapter 4, the solution is evaluated in terms of protection effectiveness and performance overhead. Finally, Chapter 5 concludes this work and discusses how it can be extended in the future.

FUNDAMENTALS 2

This chapter introduces the theoretical background required to understand this thesis. It starts by explaining the relevant concepts commonly present in modern multiprocessor computer architectures. These concepts include fundamentals of memory paging in Section 2.1 as well as memory access ordering and atomicity in Section 2.2. Subsequently, it provides an overview of typical software vulnerabilities and defense techniques in Section 2.3. Section 2.4 sheds some light onto the code generation process in the GNU Compiler Collection. Finally, Section 2.5 closes the chapter with an overview of other works related to emulation of pointer authentication and its applications.

2.1 Virtual Memory Management

Operating systems employ *virtual memory* to enforce separation between memory belonging to different processes. Virtual memory abstracts the layout of the physical memory by providing each process with its own, very large virtual address space. Virtual memory is a crucial security and safety concept, since memory isolation prevents misbehaving processes from accessing the data of other processes. Apart from that, virtual memory also allows the operating system to establish and enforce access permissions on memory regions. For instance, specific memory regions can be marked as read-only or non-executable, causing such accesses to raise an exception.

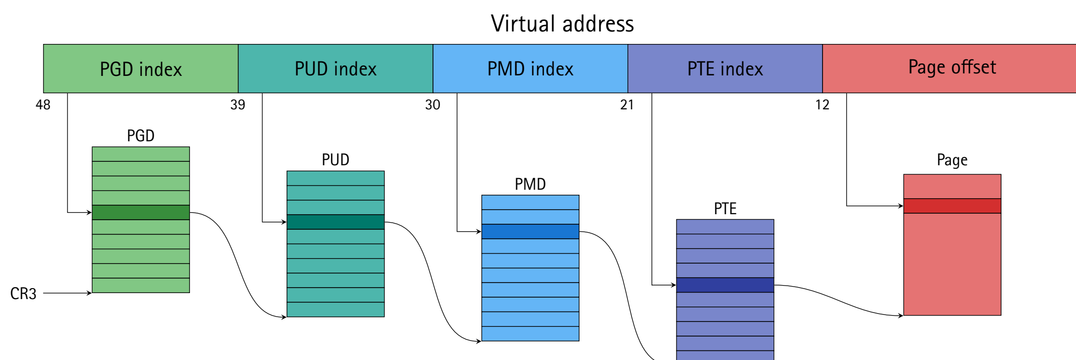


Figure 2.1 – Virtual address and the corresponding 4-level page table walk. The naming of the page table levels follows the Linux kernel convention.

Current implementations of virtual memory, including the 64-bit architectures relevant to this work, are based on *memory paging*. This technique introduces granularity into the memory management, by dividing physical memory into chunks of constant size, called *pages*. The pages are typically 4 KiB and are henceforth assumed to be of this size. Virtual address spaces of processes are established by constructing *page tables*, which store the mapping between the virtual and physical memory pages as well as the access permissions. To optimize the memory footprint, the page tables are split into multiple levels, forming a sparsely-populated tree. The different levels are indexed using parts of the virtual address and the respective entries refer to the location of the next-level page table. The last-level page table points to the 4 KiB physical page and the 12 least significant bits of the virtual address specify the offset into this page. Figure 2.1 demonstrates the composition of a virtual address and the corresponding layout of a 4-level page table.

On memory accesses, the *memory management unit* (MMU) translates the virtual address to the physical address and verifies that the access is allowed according to the page permissions. To avoid the expensive page walk for each memory access and speed up the translation, the MMU caches the recent translations in the *translation lookaside buffer* (TLB). In case of an illegal memory access or a missing translation, an exception is generated. To handle this exception, the operating system provides a *page-fault handler*. The fault handling gives the operating system the ability to load-in pages lazily, perform memory-mapped file I/O, or to create Copy-on-Write (COW) mappings. If the operating system is unable to handle the exception, it terminates the application. This reason for this condition is mostly a bug in the application that triggers an invalid memory access. However, it may also result from failed attempt to exploit a security vulnerability, as explained further in Section 2.3.

2.2 Shared-Memory Multiprocessing

Symmetric multiprocessing (SMP) systems feature several CPU cores connected to a shared main memory. This arrangement enables parallelization of multithreaded applications, which use the shared memory as the communication medium. Due to the nature of modern superscalar processing units that perform out-of-order execution, several aspects have to be considered when it comes to memory accesses.

Memory accesses are perhaps one of the slowest CPU operations. This comes from the fact that memory storage devices have been developed at a different pace than other components of the system for cost reasons [6]. The instructions operating on CPU registers are up to two orders of magnitude faster than uncached memory accesses [7]. To minimize the effect of slow memory accesses, CPUs include multiple levels of faster, albeit smaller memory. These *CPU caches* are typically implemented as an SRAM (in contrast to the DRAM main memory) and are used to cache memory accesses and transparently speed up the execution. The caches form a hierarchical structure based on their access time and size, with slower (but bigger) caches backing faster (but smaller) caches located closer to the core. While the small caches are implemented for each core separately, the high-level caches may be shared by multiple cores. To fully utilize the bandwidth of the CPU caches and the main memory, CPUs can reorder memory accesses, which do not have data or control dependencies between them. In doing so, they maintain the apparent program behavior as long as the program is single-threaded. However, the reordered memory accesses become apparent when observed from other cores of the CPU. This has to be taken into account when designing low-level multithreaded programs or lock-less data structures.

2.2.1 Memory ordering

The C++11 standard formalized the concept of *load-acquire* and *store-release* semantics [8]. According to this model, a load that is said to have acquire semantics cannot be reordered with loads and stores appearing *after* it in the program code. The store that is said to have release semantics cannot be reordered with loads and stores *before* it. This concept has been since picked up by other modern programming languages and processor architectures. The Linux kernel provides primitives for both load-acquire and store-release operations.

Table 2.1 illustrates the possible kinds of memory reordering on AArch64 and x86-64. AArch64 is an example of an architecture with the relaxed memory ordering, which provides limited guarantees about the order of memory accesses. The only memory accesses that are guaranteed to retain their order are those that have a dependency on each other [9]. However, the architecture provides plenty of means for the programmer to enforce the proper order in multithreaded environments. The load-acquire `LDAR` and store-release `STLR` instructions constitute ordered counterparts of the regular load `LDR` and store `STR` instructions. Apart from that, the architecture features a `DSB` (Data Synchronization Barrier) instruction, which waits for completion of memory accesses. The `DMB` (Data Memory Barrier) instruction is a weaker, but more efficient version of the `DSB` instruction. It enforces the relative ordering of memory accesses without waiting for their completion. Both of these instructions expect the type (loads, stores, both) and the shareability domain of the accesses in question specified as the operand.

Contrary to AArch64, the x86-64 memory model provides strong guarantees regarding memory ordering. With regard to regular memory accesses, the architecture only allows reordering of reads with older writes to different locations [10]. This rule violates neither load-acquire nor store-release semantics. Thus, any regular load has acquire semantics and any regular store has release semantics. Similarly to AArch64, the x86-64 architecture also provides memory barrier instructions to enforce the proper order of memory accesses. Just like `DMB`, the `MFENCE` instruction ensures relative ordering of memory accesses preceding it against memory accesses following it. Because of the already present guarantees, the two other x86-64 memory barrier instructions `SFENCE` and `LFENCE` are mainly limited to edge cases, which are out of scope of this section.

2.2.2 Atomicity

Implementation of concurrent algorithms often requires modifying a value in memory, such that the update would be perceived as a single *atomic* step by other observers. A simple example would be a shared counter that is being incremented by multiple threads. The incrementation of the counter in memory consists of three steps, namely: (1) loading its value from memory, (2) adding one to it, and (3) storing the new value in memory. When executed simultaneously by multiple execution threads, these steps become interleaved, resulting in a corruption of the

Type	AArch64	x86-64
Loads reordered with older loads	✓	
Loads reordered with older stores	✓	✓
Stores reordered with older loads	✓	
Stores reordered with older stores	✓	

Table 2.1 – Allowed memory access reordering on AArch64 and x86-64.

```
1 1:  ldaxr   w1, [x0]      /* Load-Acquire Exclusive */
2     add     w1, w1, #1    /* Incrementation */
3     stxr    w2, w1, [x0]  /* Store-Exclusive */
4     cbnz    w2, 1         /* Retry if store failed */
```

(a) On AArch64, using a load-exclusive and store-exclusive instruction pair.

```
1     lock                  /* Lock prefix */
2     incl    (%eax)        /* Incrementation */
```

(b) On x86-64, using the lock instruction prefix.

Figure 2.2 – Implementation of an atomic counter incrementation on different architectures.

value. The two processor architectures concerning this thesis provide instructions for performing such *read–modify–write* operations atomically, but have different approaches to the solution.

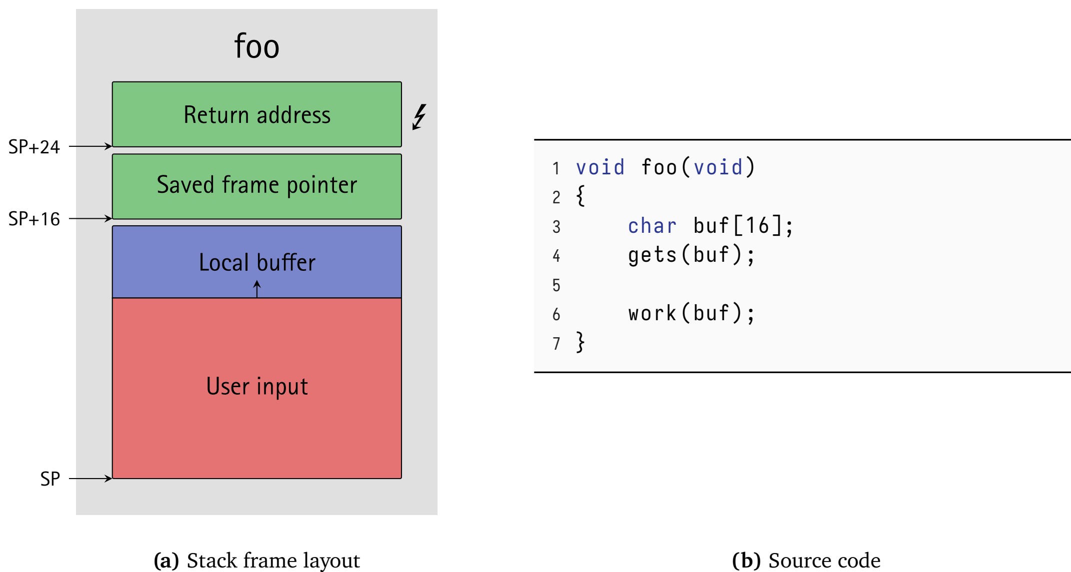
Figure 2.2 demonstrates the assembly code performing an atomic counter update on different architectures. On AArch64, the atomic operations are achieved by using a *load-exclusive* and *store-exclusive* instruction pair. The `LDXR` instruction retrieves a word from memory and marks the address in the *exclusive access monitor*. The `STXR` instruction performs a store to the memory, but only if the address in the exclusive access monitor matches the target address. The result of the instruction, i.e., whether the store took place or not, is stored in a register. Any store to this memory address performed by other CPU cores discards the address stored in the exclusive access monitor, causing the future store-exclusive instructions to fail. Putting an arbitrary modification of the value between the load-exclusive and store-exclusive instructions and looping until the store is successful achieves an atomic update of the memory. On x86-64, arithmetic and logic operations acting on memory can be turned into atomic by marking them with the `LOCK` prefix. This prefix ensures that the CPU has exclusive ownership of the cache line for the duration of the operation. While the ARMv8.1-A Large System Extension (LSE) introduced single-instruction atomic operations akin to x86-64, these instructions cannot be utilized in code that needs to be compatible with older CPUs.

2.3 Software Vulnerabilities

Necessity to manage memory manually in addition to lacking memory and type safety in low-level programming languages like C and C++ can lead to programming errors. Some of these errors give the user a possibility to modify a memory location in a way not intended by the programmer and are referred to as *memory corruption bugs*. Such bugs constitute a *security vulnerability*, since they can be used by an attacker to influence the program's behavior and gain unauthorized access to data or services running on the system.

2.3.1 Stack-based buffer overflow

One of the most representative security vulnerabilities emerges when the attacker is able to control data on the program's *call stack* (commonly called *the stack*). The call stack is used by the executing subroutines to store their private data. Each executing subroutine maintains its own



(a) Stack frame layout

(b) Source code

Figure 2.3 – Example of a subroutine containing a stack-based buffer overflow vulnerability.

stack frame accommodating the local variables, temporary buffers, and the return state. The return state includes the address of the instruction following the subroutine's call site, called *the return address*, and (optionally) the frame pointer of the caller. To finish execution, after restoring the call-preserved registers to their initial values, the subroutine issues a return instruction. This transfers the program control back to the caller by resuming execution at the return address.

Suppose the attacker is able to perform an unbounded write on the stack. In that case, the attacker can overwrite the return address of the subroutine and redirect the execution to an arbitrary location. This situation arises, for instance, when the program receives input from the user and stores it in a buffer located on the stack without validating the input size. Figure 2.3 demonstrates a vulnerable subroutine and its typical stack frame layout on x86-64 and AArch64 systems. The subroutine `foo()` allocates a 16-byte-long buffer on the stack to store the user input. After that, the buffer address is passed to the infamous `gets()` [11] function of the C standard library, which reads a line from the standard input into the buffer. Since `gets()` does not perform any bounds checking, a malicious user can overwrite adjacent data on the stack by providing an input that is longer than 15 characters. In this example, an input string containing 16 characters would cause the null-terminator character to overwrite a part of the saved frame pointer. Hence, a 32-character-long input string gives the attacker complete control over the return address and the program's execution flow.

It is worth noting that the example provided in Figure 2.3 is intentionally oversimplified for brevity purposes. The memory corruption bug in `foo()` could be easily detected by static analysis and fixed by replacing `gets()` with a function that performs bounds checking. In reality, such vulnerabilities are typically hidden behind manual pointer manipulation or explicit loops and span over abstraction layers [2]. This issue raises the need for systematic approaches to prevent the exploitation of software vulnerabilities.

2.3.2 Conventional mitigations

Due to the increasing complexity of present software, manual attempts to remove software vulnerabilities at the development stage rarely lead to anticipated results. To make matters worse, these attempts might give a false sense of security or even introduce new bugs [2]. Unfortunately, state-of-the-art static analysis tools are likewise unable to detect complex security vulnerabilities [12]. That leads to the use of various defense techniques, which aim to protect programs by preventing the exploitation of vulnerabilities at runtime.

In the earlier days of computing, commodity systems lacked certain memory protection features. The attackers would exploit stack-based buffer overflows by placing the malicious payload (also called *shellcode*) directly on the stack and redirecting the control flow to it [13]. Modern hardware has since rendered this exploitation technique useless by allowing more extensive control over the virtual memory access rights. In particular, implementing the Write xor eXecute (W⊕ X) memory protection policy in operating systems has become the industry standard [14, 15]. W⊕ X dictates that any region in memory must be either writable or executable, but never both. This limitation considerably complicates the exploitation of stack-based buffer overflow vulnerabilities. An attacker who managed to place the shellcode in a writable memory region (such as the stack) will not be able to execute it. Analogously, it is impossible to introduce new code into the executable regions. Unfortunately, applications relying on self-modifying code or Just-In-Time (JIT) compilation cannot enforce W⊕ X fully. This is a concern for present-day mainstream web browsers, all of which include a JIT compiler to speed up JavaScript execution.

The inability to smuggle malicious code into the program memory has led to the discovery of several new exploitation paradigms. Instead of injecting any code, the *code-reuse attack*s (CRAs) construct the payload from already existing executable code in the program. The first code-reuse attack *return-into-libc*, presented by Peslyak [16], circumvented W⊕ X by making

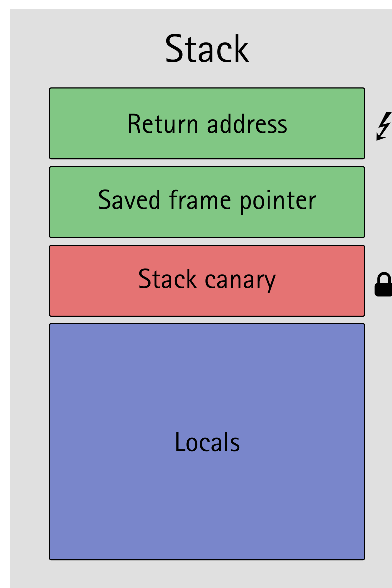


Figure 2.4 – Stack frame protected with a stack canary.

the vulnerable subroutine return into a C standard library (*libc*) function, allowing him to execute the system shell. Peslyak also showed that it is possible to chain functions in some cases, allowing the attacker to elevate permissions before executing the shell. Shacham [17] took this concept even further and introduced *return-oriented programming* (ROP). ROP attacks combine short instruction sequences ending with a return instruction into basic building blocks called gadgets. Given enough executable code in the program memory, the attacker can construct a Turing-complete gadget catalog including gadgets for arithmetic, logic, function calls, loops, and conditional branches. The attacker is then able to perform arbitrary computations by chaining these gadgets. Roemer et al. [18] formalized this concept and developed an exploit language and a compiler, simplifying creation of return-oriented programs even more.

Modern operating systems generally employ, among other things, *address-space layout randomization* (ASLR) [19] to combat CRAs. ASLR is a simple and effective technique that prevents the attacker from reliably exploiting code in memory by arranging code areas randomly on each run. This significantly complicates the return-into-library attacks or construction of ROP chains, requiring the attacker to guess the location of the exploitable code. However, ASLR is susceptible to memory corruption bugs that disclose information which is not intended to be readable by the user [20]. Malicious actors can use these *information leaks* to revert the randomization and reconstruct the layout of the program code. Szekeres et al. [3] show that most memory corruption bugs can be exploited in a way that causes information leakage.

2.3.3 Control-Flow Integrity techniques

In contrast to W⊕ X and ASLR, which are enforced by the operating system, the *Control-Flow Integrity* (CFI) techniques involve embedding the protection into the program's machine code. Some of these techniques have found their way into modern compilers and seen wide adoption in security-critical applications. One of the most prominent techniques is *stack-smashing protection* (SSP), implemented by GCC's `-fstack-protector` flag [21]. It protects the return address of the subroutine by placing a secret value called *stack canary* on the stack frame. The program initialization code generates a random reference value for stack canaries and stores it in a global variable. This value is then placed during stack frame allocation in such a way that a buffer overflow targeting the return address would overwrite it. The compiler augments the code of the subroutine to compare the value of the stack canary with the reference value before returning. If the value has been tampered with, the program terminates execution. Figure 2.4 illustrates an example of a stack frame protected with a stack canary. Since standard library functions operating on null-terminated strings stop at the zero byte, the protection can also be hardened by including a zero byte in the canary value [22].

To mitigate the additional overhead introduced by the comparison, GCC uses heuristics to omit functions that are unlikely to contain security vulnerabilities from protection. The default policy enforced by `-fstack-protector` is to protect only the functions that call `alloca()` compiler builtin or that contain a character buffer of size `ssp-buffer-size` or larger. The value of `ssp-buffer-size` parameter can be tweaked using compiler flags and defaults to 8 in most distributions. The flag `-fstack-protector-all` protects all functions in the program. The Chrome OS team was using this option due to security concerns but was not satisfied with the performance penalty [23]. For that reason, another policy, enforced by `-fstack-protector-strong` was introduced. It protects functions that contain arrays of arbitrary type and length, calls to `alloca`, and variables that had their address taken.

Stack canaries offer protection primarily against contiguous stack buffer overflows. Some memory corruption bugs give the attacker the ability to write an arbitrary value to an arbitrary

location. These memory bugs, also known as *write-what-where* conditions, empower the attacker with the ability to overwrite the return address without touching the stack canary. Alternatively, having determined the location of the reference value, the attacker can either set it to a custom value or leak it. The attacker can then perform a contiguous buffer overflow, while preserving the correct canary value. In any of these cases, the canary comparison would neither detect the value mismatch nor terminate the program, resulting in a successful control-flow redirection.

Some protection techniques offer a fine-grained highly effective CFI, but suffer from significant performance overhead [3]. *AddressSanitizer* [24] uses a region in memory, called *shadow memory*, to mark valid memory access targets at run time. This technique is implemented in major compilers and detects a wide range of memory corruption bugs at the cost of up to 2x memory and performance overhead. Label-based CFI techniques instrument indirect jumps to only allow valid targets identified as such by static analysis [25]. On indirect jump, the target of the indirect jump is compared with the control-flow graph. However, due to limitations of static analysis, the label-based CFI techniques fall back to overly permissive choices or simplify runtime checks to remain practical [26].

2.3.4 ARMv8.3-A pointer authentication

In 2017, the ARMv8.3-A architecture introduced a pointer authentication mechanism implemented in hardware [4]. This mechanism implements efficient protection of code and data pointers with negligible performance overhead and memory footprint. The length of the virtual addresses on AArch64 is subject to system configuration and ranges between 32 and 52 bits, with another bit being used to toggle between low and high halves of the address space [9]. Typical virtual memory configurations using 4 KiB pages feature 48-bit virtual addresses. Despite this, the pointers are generally treated as 64-bit values due to the architecture's word size and the unused upper bits contain the sign extension of the virtual address. Depending on the size of the address space and whether the address tagging feature is enabled, the amount of unused bits ranges from 3 to 31 bits. These unused bits of the pointer can be used to store the cryptographic signature of the pointer.

The pointer authentication mechanism offers a set of instructions for creation and validation of such signatures. The instructions under the mnemonic PAC<key> <pointer>, <context> augment the pointer stored in the first operand register with a cryptographic signature called *pointer authentication code* (PAC). Figure 2.5 illustrates the operation of the PAC instructions. The computed PAC includes the value stored in the <context> register as the hashing salt and the <key> placeholder in the instruction mnemonic indicates the used key. The mechanism features five different 128-bit keys: two instruction keys IA and IB, two data keys DA and DB, and a general-

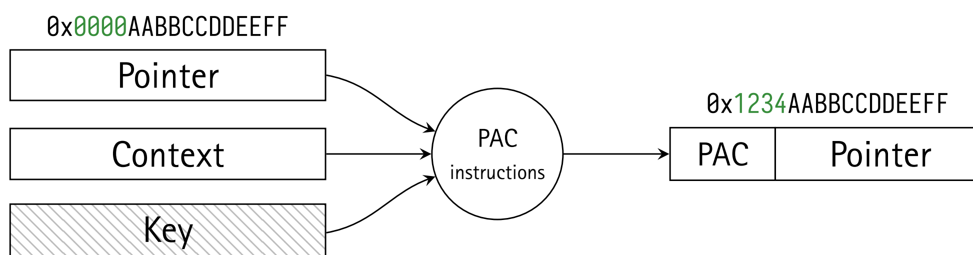


Figure 2.5 – Functional principle of the PAC instructions.

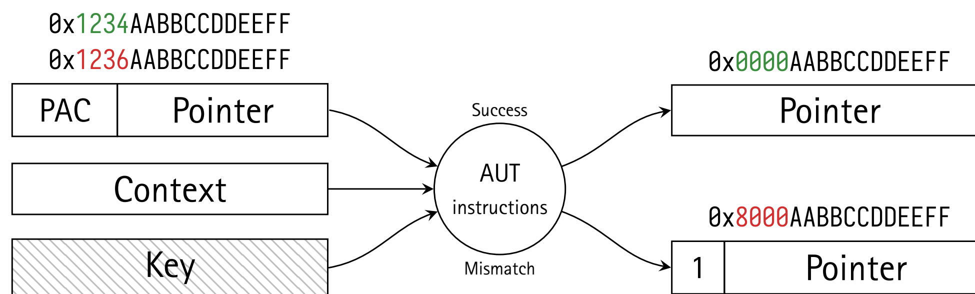


Figure 2.6 – Functional principle of the AUT instructions.

purpose key GA. The keys are stored in the system control registers and are not accessible from the userspace. Instead, the code running at higher privilege levels, such as the operating system or the system hypervisor, is responsible for their management. The QARMA block cipher [27] was developed specifically to be used for PAC computations. This cipher is resistant to truncation and is meant to be deployed in fully unrolled or pipelined hardware implementations.

A set of symmetric instructions following the `AUT<key> <pointer>, <context>` mnemonic performs the reverse operation. These instructions compute the PAC and compare it with the value already stored in the `<pointer>` register. If the register contains the result of the respective PAC instruction performed with the same context value, the values will match. In this case, the instruction restores the original pointer value by removing the signature from the upper bits. If the authenticated pointer has been tampered with (e.g., as a result of an attack), the comparison detects a mismatch and the instruction corrupts the upper bits of the pointer. In practice, this often means setting the most significant bit of the pointer to one. Dereferencing such a pointer would generate a translation fault, causing the operating system to terminate the application. Figure 2.6 illustrates the operation of the AUT instructions. The pointer authentication mechanism also provides XPAC instructions for stripping the PAC without validation.

The subroutine calls on AArch64 are done by issuing the BL instruction. This instruction puts the address of the following instruction (PC+4) into the link register (LR) and branches to the target address in the operand. Respectively, the return instruction RET transfers the control flow back to the caller by branching to the address in the LR. Since nested calls would overwrite the LR, non-leaf functions save the LR on the stack in the function prologue and restore it in the function epilogue. This yields the stack arrangement introduced in Section 2.3.1, which is vulnerable to

```
1  paciasp                               /* Insert signature into LR */
2  stp     fp, lr, [sp, #-FRAME_SIZE]!   /* Save LR, FP on the stack */
3  mov     fp, sp                        /* Set the new frame pointer */
4
5  /* Function body */
6
7  ldp     fp, lr, [sp], #FRAME_SIZE     /* Restore LR and FP */
8  autiasp                               /* Verify integrity of LR */
9  ret
```

Figure 2.7 – Return address protection using the ARM pointer authentication.

stack-based buffer overflows. The pointer authentication mechanism provides effective means for protection of the return address stored on the stack. The specialized instructions `PACIASP` and `AUTIASP` perform authentication of the link register (LR) using the instruction key A and the stack pointer (SP) as the context value. As demonstrated in Figure 2.7, protection of the return address is the matter of issuing `PACIASP` before pushing the return address onto the stack and `AUTIASP` immediately after popping it of the stack. These specialized instructions are located in the NOP-space to ensure binary compatibility with older versions of the architecture.

Applications of the ARMv8.3-A pointer authentication mechanism are not limited to protection of return addresses. Another obvious use case is the protection of writable function pointers and pointers to the C++ virtual method tables (vtables). The naming scheme of the keys also suggests using it for protection of sensitive data pointers. Furthermore, the general-purpose `PACGA` instruction computes a 32-bit PAC from two 64-bit operands. This instruction can be used to create authenticated buffer canaries or it can be chained to protect arbitrary-sized data [28]. Note that there is no `AUTGA` counterpart to the `PACGA` instruction. The validation is performed by issuing another `PACGA` instruction and comparing the result manually. Section 2.5.3 provides an extended overview of CFI techniques based on the pointer authentication mechanism.

The main attack vector associated with this mechanism is that it might be possible to brute-force particularly short PACs [4]. This issue is worsened by the nature of the `fork()` system call on Unix systems. Given that the new process is a complete duplicate of the parent process, it also retains the original key. While constructing such an attack is not trivial, an attacker who is able to issue the `fork()` system call can test forged pointers without crashing the original process. In June 2022, Ravichandran et al. [29] presented a side-channel attack targeting the pointer authentication on Apple's M1 CPUs. The attack works by examining the microarchitectural side effects of a speculatively executed gadget, which consists of authentication and dereference of a signed pointer, on the CPU caches. They were able to observe that only a correct PAC causes the CPU to evict TLB entries. Since the architectural effects of speculative execution are not committed, the attacker is able to brute-force the PAC without crashing the process or being detected.

2.4 Code Generation in GCC

Many protection techniques embed additional code into the program and add checks, which ensure the integrity of the execution flow. This work is not an exception and relies on a plugin for the *GNU Compiler Collection* (GCC), which performs code instrumentation to protect return addresses on the stack.

The GCC is an optimizing compiler developed as a part of the GNU Project. While the compiler was originally developed for C, it has gained support for several programming languages, namely C++, Objective-C, Objective-C++, Fortran, Ada, D, and Go, as well as dozens of hardware architectures [21]. To achieve this, GCC splits the translation process into multiple passes, performed by the front-end, middle-and, and back-end of the compiler [30]. The front-end deals with language-specific aspects of the compilation and generates an intermediate representation, called *GIMPLE*, which is then fed into the middle-end. The middle-end is both language- and architecture-agnostic and applies high-level abstract optimizations, including dead code elimination and function inlining. The back-end works on the *register transfer language* (RTL) representation of the program and consists of multiple passes. The RTL is inspired by Lisp lists and describes instructions along with their side effects. The back-end starts with a hardware-independent RTL with unlimited register set, which is then iteratively optimized and

adapted to the target architecture using pattern matching. For example, the `ira` (Integrated Register Allocator) pass replaces the virtual registers by allocating platform registers of the target CPU. The end product of the back-end is the RTL representation of the assembly instructions in the compiled program, which can easily be translated into the machine code. In this way, the programming language front-ends benefit from the wide variety of architectures and optimizations supported by the GCC middle-end and the architecture back-ends.

The version 4.5.0 of the GCC added support for plugins [31]. The plugins are loaded as shared libraries and allow extending the compiler without modifying its source code and requiring a complete rebuild. The GCC provides an API for the plugin developers to add additional passes or provide callbacks for certain events. This can be used to introduce new optimizations and perform code transformation or analysis based on information available only during build-time.

2.5 Related Work

Pointer authentication is not the only hardware-based mechanism that enforces integrity of return addresses. Various hardware-assisted techniques by security researchers and CPU manufacturers present diverse approaches to return address protection or emulation of the original ARMv8.3-A mechanism. Furthermore, prior to the introduction of the ARMv8.3-A pointer authentication mechanism, multiple works proposed software-based techniques that protect code and data pointers from control-flow redirection attacks in a similar fashion.

2.5.1 Software-based protection of code pointers

The idea of adding a cryptographic signature to the pointers is not new. In 2015, Mashtizadeh et al. [26] proposed adding message authentication codes (MACs) to control-flow elements, such as return addresses, function pointers, and vtable pointers in a technique called *CCFI*. To avoid information leaks disclosing the secret key, CCFI reserves 11 XMM registers to store the key. Since this constitutes a change to the application binary interface (ABI), all the dependencies of the program require recompilation. The *Code-Pointer Integrity* (CPI) approach [32] aims to protect pointers by storing them in an isolated memory location along with some metadata. On AArch64 and x86-64 this isolation relies on the fact that no addresses pointing into the isolated region are ever stored in the regular memory. On x86-64 this is achieved by storing the address of the isolated region in one of the unused segment registers. Evans et al. [33] presented an attack that is able to bypass CPI on these architectures and argued that security mechanisms relying on information hiding are ineffective.

2.5.2 Hardware-based techniques

This thesis builds upon the work of Serra et al. [5] introducing *PAC-PL*. It presents a hardware-assisted emulation of the ARMv8.3-A pointer authentication mechanism on systems that include a field-programmable gate array (FPGA). PAC-PL comes with OS and compiler support and also explores advanced key management and attack detection strategies. The evaluation shows moderate performance overhead, mitigated by protecting only functions that allocate arrays on the stack. Another work exploring hardware-assisted pointer authentication is *RetTag* [34], which expands the RISC-V architecture with a mechanism imitating the ARMv8.3-A pointer authentication. The prototype of RetTag is implemented and integrated into a FPGA-synthetized RISC-V core.

Pointer authentication is not the only hardware-assisted method to protect backward branches. A 2017 survey of hardware-based CFI techniques [35] provides an extensive overview of works proposing protection techniques implemented in hardware. It concludes that most presented techniques protect the return addresses of subroutines by using a *shadow call stack*, an additional hardware stack used to store an extra copy of the return address. This also applies to the *Control-Flow Enforcement Technology* [36], a hardware-based CFI technique developed by Intel and already present on some recent CPUs. Apart from providing protection of backward branches, the Control-Flow Enforcement Technology protects forward branches using a technique called *Branch Limitation*, which allows indirect branching only to valid entry points marked with `ENDBRANCH` instruction. The ARMv8.5-A architecture introduced a similar mechanism called *Branch Target Identification* (BTI) [9]. Christou et al. [37] implemented the shadow call stack mechanism for SPARC V8 architecture by extending the instruction set of a processor core synthetized on an FPGA.

2.5.3 Applications of pointer authentication

Return address protection is by far not the only application of the pointer authentication mechanism. In the recent years, many research projects have proposed different CFI techniques that are based on PAC codes. Considering the sparsity of ARMv8.3-A systems with support of the PAC mechanism, many of these works can benefit from software-emulated pointer authentication mechanism presented in this thesis.

Liljestrand et al. presented several works utilizing the pointer authentication for novel CFI techniques. *PARTS* [28] is a compiler instrumentation framework, which protects a wide range of pointers with PAC codes. This includes return addresses, local, global and static pointers, and pointers in C structures. To facilitate run-time type safety for data and code pointers, the context value for the PAC generation includes the information about the type of the pointee. The performance evaluation shows 19.5 % average overhead for data-pointer signing. *PCan* [38] is an improved mechanism for SSP stack canaries, which does not involve storing the reference value in memory. Instead, the context-based canary value is dynamically generated for each function call using the `PAC` instructions. *PACStack* [39] builds the context value for return address signature by cryptographically binding it to all previous return addresses in the call stack. This technique makes signed pointers unique to a particular control-flow path and thus prevents pointer-reuse attacks with minimal performance overhead (ca. 3 %).

Farkhani, Ahmadi, and Lu designed *PTAuth* [40], a scheme to detect temporal memory corruptions (i.e., use-after-free, double-free, invalid-free errors) at runtime. As a proof of concept, the evaluation used a simple software-emulated pointer authentication with PAC computed by XORing the context value and the pointer. Backed by such emulation, PTAuth shows a moderate performance slowdown of 26 %. Denis-Courmont et al. [41] demonstrated how the Linux kernel security can benefit from the pointer authentication mechanism by proposing a range of use-cases and key management mechanisms. However, since the software-emulated pointer authentication mechanism presented by this thesis relies on OS support to provide the pointer authentication, it cannot be used to protect the OS kernel itself.

2.6 Summary

The complexity of present computer software raises the need for protecting it from malicious actors. A prominent protection technique is the pointer authentication mechanism introduced by

the ARMv8.3-A architecture. The mechanism allows enforcing the integrity of the execution flow with very low performance overhead thanks to its hardware-based implementation. The research in the software security field has already presented many techniques that put this mechanism to use. Due to limited availability of systems which include this mechanism, a software-emulated solution is developed and presented within the scope of this thesis. This solution, explained in-depth in the following chapter, is integrated into the operating system and makes use of the virtual memory mechanism of x86-64 and AArch64 processor architectures. Since it involves communication between several CPU cores, the critical code sections have to be designed with the aspects of memory ordering and atomicity in mind.

3 ARCHITECTURE

This work presents a complete Control-Flow Integrity (CFI) solution for return address protection and consists of two separate components. The first component is the *kpac* Linux kernel extension, which provides a pointer authentication service for applications running in userspace. The second component is the accompanying GCC plugin called *PAC-SW*, which augments the code of the userspace applications with a return address protection technique facilitated by the kpac pointer authentication service. This chapter starts with a high-level overview of the communication protocol between the operating system and the userspace applications in Section 3.1. After that, it introduces the design and the internals of the PAC-SW plugin in Section 3.2. In the end, Section 3.3 provides an insight into the inner workings of the kpac kernel extension as well as its configuration options.

3.1 Communication Protocol

The main idea of the technique presented in this thesis is to provide an OS-based pointer authentication mechanism that does not involve system calls or page faults. This goal is motivated by the significant performance cost induced by switching into the privileged execution mode of the OS kernel. As demonstrated by Serra et al. [5], performing a system call or handling a page fault in each function prologue and epilogue would multiply this cost and render the technique unusable. As an alternative, this work proposes to use a shared memory page as the communication medium between the kernel extension and the userspace applications running on different CPU cores. To ensure fast response of the pointer authentication service, the OS dedicates one or several CPU cores to await and process such pointer authentication requests. The kernel extension sets up the shared memory page (hereinafter *kpac page*) at a fixed address in the virtual address space of every userspace application. To accomplish synchronization between the pointer authentication service and the userspace applications, both perform *polling* (repeated sampling) of the kpac page. For the pointer authentication service, this means repeatedly querying the kpac page for pending pointer authentication requests. The userspace applications, in turn, submit these requests and wait for their completion by repeatedly querying their status from the kpac page.

The kpac page contains four 64-bit words, which are used as registers in the communication process, in the following order: `STATUS`, `PLAIN`, `TWEAK`, and `CIPHER`. The requests follow the same procedure on both architectures supported by the kpac pointer authentication mechanism. To perform an equivalent of the ARMv8.3-A PAC or AUT instructions, the application writes the context value and the pointer into the respective registers of the kpac page. This means writing

Register	Offset [B]	Usage during PAC requests	Usage during AUT requests
STATUS	0	Constant value: 0x1	Constant value: 0x2
PLAIN	8	Input: Unsigned pointer	Output: Unsigned/corrupted pointer
TWEAK	16	Input: Context value	Input: Context value
CIPHER	24	Output: Signed pointer	Input: Signed pointer

Table 3.1 – Registers in the kpac page and their usage during kpac requests.

the unsigned pointer into the PLAIN register for a PAC request and writing the signed pointer into the CIPHER register for an AUT request. Regardless of the request, the context value is written into the TWEAK register. Then, the request is submitted by writing either 0x1 for PAC or 0x2 for AUT into the STATUS register. This causes the pointer authentication service to fetch the inputs from the kpac page, perform the requested operation, store the output into the respective register, and set the STATUS register to zero. The output register is CIPHER in case of a PAC request and PLAIN in case of an AUT request. After submitting the request, the application repeatedly polls the value of the STATUS register. Once the value is zero, the application is free to fetch the result and continue the further execution. Thus, each register fulfills a specific purpose in the communication protocol and their usage is summarized in Table 3.1.

The cost of the memory accesses introduced in Section 2.2 raises the question whether this approach is optimal, since it involves three stores per authentication request. An early prototype of the mechanism without the STATUS register that involved only two stores showed no significant improvement in performance over the four-register version. This finding can be attributed to the fact that all the registers fit into a single cache line, which becomes exclusive to the CPU that has performed the most recent store to it. The following stores operate on the said cache line and introduce very limited overhead. Furthermore, the three-register prototype assumed that the pointers cannot be null and initiated the operation as soon as a non-null pointer was stored in the pointer register. However, there is nothing preventing the applications from authenticating null pointers in the ARMv8.3-A pointer authentication mechanism. Therefore, a communication protocol with four registers was chosen in the name of simplicity and to remain true to the original design. The presence of the STATUS register also allows extending the mechanism in the future with additional operations, e.g., to allow multiple keys.

3.2 GCC Plugin

Similar to the ARMv8.3-A pointer authentication mechanism demonstrated in Figure 2.7, enabling return address authentication in userspace applications requires additional code to be generated in the prologue and epilogue of each subroutine. The code that is inserted before the original prologue performs an equivalent of the paciasp instruction, that is, adding the cryptographic signature to the return address. The code that is inserted after the original epilogue performs an equivalent of the autiasp instruction, which means verifying the integrity of the return address by authenticating the signature stored within the unused bits of the pointer. Thus, a plugin for GCC 10.2 was developed to automate instrumentation of the subroutines during the compilation process.

3.2.1 Return address protection on AArch64

Figure 3.1 demonstrates the code that is generated in subroutines to enable return address protection on AArch64. It is noteworthy that both the epilogue and the prologue follow a similar scheme with minor differences. This is consistent with the similarities of PAC and AUT requests introduced in Section 3.1. The following paragraphs explain the process of adding a signature to the return address as part of the subroutine's prologue on AArch64 (Figure 3.1a).

Performing a request to the pointer authentication service requires two temporary registers: one that holds the address of the kpac page, and a scratch register for the values that are loaded and stored. On AArch64, this purpose is fulfilled by the registers `x9` and `x10`, respectively. According to the AArch64 application binary interface (ABI) [42], these registers are caller-saved and are not utilized otherwise in the calling convention. Thus, assuming no compiler optimizations invalidating the calling convention, their value is undefined and they can be freely used in both the prologue and the epilogue. The preparation of the request starts by storing the address of the kpac page, which is defined as the `PAC_BASE` macro, in the `x9` register (line 1). Since the stack pointer (SP) cannot be used directly in the store instructions, it is first transferred to the scratch register `x10` (line 2). The following instruction `stp` (Store Pair) stores values of two 64-bit registers sequentially in the memory, which are, in this case, `LR` and `x10` (line 3). The instruction is issued with the target address of `x9` plus an offset of eight bytes, resulting in a store to the `PLAIN` and `TWEAK` registers. The following two instructions initiate the PAC request by transferring `0x1` to the scratch register (line 4) and storing it with release semantics into the `STATUS` register of the kpac page (line 5). Next, the `STATUS` register is polled repeatedly until its value changes to zero.

To achieve limited power consumption and bus contention during polling, the loop uses the *Wait for Event* mechanism of AArch64. The `wfe` instruction (line 8) indicates that the CPU core can suspend execution and enter a low-power state until it is woken up by another CPU core. The wakeup event occurs when another CPU core issues a `sev` (Send Event) instruction or when a store from another CPU core clears the local exclusive access monitor. The `wfe` instruction is preceded by a `sevl` (Send Event Local) instruction (line 7). This instruction is issued to prepopulate the event queue, so that the `wfe` instruction acts as a NOP on the first iteration of the loop. Within the loop, the `ldxr` (Load Exclusive Register) instruction loads the value from the `STATUS` register (line 9). If its value is not zero, the `cbnz` (Compare and Branch If Not Zero) jumps

```
1      mov    x9, #PAC_BASE
2      mov    x10, sp
3      stp    lr, x10, [x9, #8]
4      mov    x10, #1
5      stlr   x10, [x9]
6
7      sevl
8  1:  wfe
9      ldxr   x10, [x9]
10     cbnz   x10, 1
11     ldr    lr, [x9, #24]
```

(a) Before the function prologue.

```
1      mov    x9, #PAC_BASE
2      mov    x10, sp
3      stp    x10, lr, [x9, #16]
4      mov    x10, #2
5      stlr   x10, [x9]
6
7      sevl
8  1:  wfe
9      ldxr   x10, [x9]
10     cbnz   x10, 1
11     ldr    lr, [x9, #8]
```

(b) After the function epilogue.

Figure 3.1 – Additional code generated in functions for return address protection on AArch64.

back to wfe (line 10). Note the missing stxr counterpart to the ldxr instruction. In this case, the load-exclusive mechanism is used not to achieve atomic memory update, but to receive a wakeup event once another CPU core performs a store to the STATUS register. Once the STATUS register changes its value to zero, the loop is exited. The following ldr (Load Register) instruction loads the signed pointer from the CIPHER register of the kpac page into the LR register of the CPU (line 11).

3.2.2 Return address protection on x86-64

Figure 3.2 demonstrates the code that is generated in subroutines to enable return address protection on x86-64. The rest of this section describes the process of authenticating the signature of the return address in the subroutine's epilogue (Figure 3.2b).

Also in this case two temporary registers are required to hold the address of the kpac page and the manipulated values. The x86-64 calling convention on Linux specifies registers r10 and r11 as the only caller-saved general-purpose registers [43]. In fact, the register r10 does have a purpose of passing the function's static chain pointer, but this feature is not used by most programming languages, including C. The preparation of the request begins by storing the address of the kpac page in r10 (line 1) and copying the signed return address from the top of the stack into the r11 register (line 2). Then, both the return address in r11 and the stack pointer in rsp are stored into their respective kpac registers CIPHER and TWEAK (lines 3–4). Finally, the AUT operation is initiated by storing 0x2 into the STATUS register (line 5).

Unfortunately, the x86-64 architecture does not offer a way to perform efficient polling in userspace akin to the WFE mechanism of AArch64. The value of the STATUS register is continuously read from memory and compared to zero using a single cmpq instruction (line 7). If the comparison is successful, the loop is exited (line 8). Otherwise, a pause instruction followed by an unconditional jump back to the comparison are issued (lines 9–10). The pause instruction, also known as a nop with a rep prefix, improves performance of such spin-wait loops [10]. It prevents the CPU from speculatively executing the comparison or predicting its result, which would introduce a performance penalty once the value of the STATUS register changes. After the loop is exited, the unsigned pointer in the PLAIN kpac register is transferred to r11 (line 11), which is then written to the top of the stack (line 12), overwriting the signed pointer.

```
1       movq    $PAC_BASE, %r10
2       movq    (%rsp), %r11
3       movq    %r11, 8(%r10)
4       movq    %rsp, 16(%r10)
5       movq    $1, (%r10)
6
7   1:  cmpq    $0, (%r10)
8       je      2
9       pause
10      jmp     1
11  2:  movq    24(%r10), %r11
12      movq    %r11, (%rsp)
```

(a) Before the function prologue.

```
1       movq    $PAC_BASE, %r10
2       movq    (%rsp), %r11
3       movq    %r11, 24(%r10)
4       movq    %rsp, 16(%r10)
5       movq    $2, (%r10)
6
7   1:  cmpq    $0, (%r10)
8       je      2
9       pause
10      jmp     1
11  2:  movq    8(%r10), %r11
12      movq    %r11, (%rsp)
```

(b) After the function epilogue.

Figure 3.2 – Additional code generated in functions for return address protection on x86-64.

3.2.3 Configuration options

Before diving into the implementation details of the PAC-SW plugin, it is reasonable to inspect it from the user's perspective first. Despite avoiding overhead of context switching, the software-emulated pointer authentication mechanism still introduces significant overhead into protected programs when applied carelessly. This is due to relying on shared memory to facilitate communication between different CPU cores. The CPU caches are able to conceal the cost of memory accesses for locations accessed primarily by a single CPU core. However, if the location is accessed concurrently by multiple cores, the CPU is required to maintain *cache coherency*, or consistency of the data in caches across different cores. This comes with a performance cost.

To mitigate the overhead introduced by the pointer authentication, the GNU Compiler Collection (GCC) plugin offers several heuristics to omit functions that are unlikely to contain stack-based buffer-overflow vulnerability from protection. These heuristics are called *protection scopes* (strategies) and can be specified using the command-line argument `scope`. Table 3.2 lists the protection heuristics that are implemented in the PAC-SW plugin. The `nil` and `all` protection scopes can be used to disable or enable protection in all functions. This is useful in combination with the `pac_scope` function attribute, which forces specific strategy during the compilation of the annotated function, as shown in Figure 3.3.

Another mitigation of the performance overhead is possible on AArch64. Since the return address is initially stored in the link register (LR) and *leaf functions* do not call other functions, they need not save their return address on the stack. Therefore, a stack-based buffer overflow is not possible and, by default, no return address protection is added. This behavior can be overridden by providing a `leaf` command-line argument (possible values: `y`, `n`).

Other accepted command-line arguments are `dump` and `init`. The `dump=<filename>` argument causes the plugin to output the list of functions that were complemented with return address protection into a file and was used during evaluation of the technique. The `init=<function>` argument allows specifying a function that is called during program initialization to set up the return address protection. While such function was required for early prototypes of the technique, operating system support eliminated this need and rendered this argument redundant.

3.2.4 Compatibility with compiler optimizations

This thesis aims to improve the software infrastructure developed for PAC-PL [5]. One drawback of the original GCC plugin is the missing support for compiler optimizations, which constitutes a hurdle to its application. Therefore, a goal was set during the development of PAC-SW to add such support. In practice, this involved identifying and automatically disabling incompatible

Functions containing...	`nil`	`char`	`array`	`strong`	`all`
calls to `alloca()` compiler builtin		✓	✓	✓	✓
char arrays of size ≥ `ssp-buffer-size`		✓	✓	✓	✓
arrays of arbitrary size and type			✓	✓	✓
variables on the stack that had their address taken				✓	✓
All functions					✓

Table 3.2 – Protection scopes implemented by the PAC-SW plugin.

```
1 __attribute__((pac_scope("nil"))) int foo(void) { /* ... */ }
2 __attribute__((pac_scope("all"))) int bar(void) { /* ... */ }
```

Figure 3.3 – Exemplary usage of the `pac_scope` attribute.

optimizations, while remaining compatible with as many optimizations as possible. Thus, following optimizations are automatically disabled by the plugin:

`-fipa-ra`: This optimization enables the compiler to omit saving caller-saved register before calls if it is able to determine that the called function does not clobber them [21]. Such optimization is incompatible with the return address protection, because the authentication code clobbers the caller-saved registers, namely `r10`, `r11` on x86-64 and `x9`, `x10` on AArch64.

`-fshrink-wrap`: This optimization enables the compiler to delay function prologue in case it is not needed for some code paths [21]. An example of such case would be a subroutine that immediately returns if one of its arguments is null. The main issue caused by this optimization is that the code before the frame setup and pointer authentication can store some values into the caller-saved registers. In case these registers are accessed again after the function prologue, their values are clobbered by the pointer authentication code.

`-freorder-blocks`, `-freorder-blocks-and-partition`: These optimizations cause the compiler to reorder basic blocks within the functions in order to reduce the number of taken branches and improve code locality [21]. In practice, this causes the compiler to generate multiple epilogues in the functions. While the plugin is compatible with multiple epilogues and exit points, the value introduced by these optimizations after the code instrumentation is questionable. This is due to the size of the authentication code presented in Sections 3.2.1 and 3.2.2. Multiplied by the amount of epilogues, the additional code causes the binary size to inflate significantly.

It is worth noting that there are other compiler optimizations that generate multiple exit paths with separate epilogues, like `-foptimize-sibling-calls`. However, these optimizations generate a low amount of additional epilogues compared to block reordering and thus have limited impact on the binary size.

3.2.5 Code instrumentation

In order to insert the code introduced in the previous section into the function prologues and epilogues, the GCC plugin operates on the register transfer language (RTL) representation of the program. This is done by registering an additional compilation pass (named `inst_pac`). The new pass is inserted after the `free_cfg` compilation pass, which is the last one to change the control-flow graph (CFG) of the program. At this stage, the stack frame layout is final and all the function prologues and epilogues are already generated.

The pointer authentication pass `inst_pac` inserts the return address protection code into the RTL by creating an inline assembly block. This inline assembly block is equivalent to the code that GCC generates for an `asm volatile` statement and is demonstrated in Figure 3.4. The `insn` expression represents a single instruction that does not jump or call a function. It possesses a unique ID (first argument) and is doubly-linked with the previous (second argument) and the next (third argument) expressions in the RTL representation of the execution flow. The effect of the `insn` expression is represented in the fourth argument, which in this case is `parallel`. The

```
1 (insn 44 39 40 (parallel [
2             (asm_input/v ("assembly code") example.c:16)
3             (clobber (mem:BLK (scratch) [0  A8]))
4             (clobber (reg:CC 17 flags))
5         ]) "example.c":16:1 -1
6      (nil))
```

Figure 3.4 – The RTL representation of an inline assembly block.

parallel expressions represents several simultaneous side effects specified in its single vector argument. The side effects of the inline assembly block are as follows:

- The asm_input side effect with volatile flag /v specifying the inline assembly code.
- The clobber side effect with (mem:BLK (scratch)) argument representing that all memory locations must be presumed clobbered. This prevents the compiler from caching variables in registers.
- The clobber side effect with (reg:CC 17) argument representing that the x86-64 flags status register must be presumed clobbered. This side effect is not present on AArch64.

Apart from these side effects, the RTL representation in Figure 3.4 also contains debugging information, such as the corresponding location in the source file, and the intermediate data for further code generation passes.

Before instrumenting the function, the inst_pac pass determines whether the function is contained in the current protection scope. This is done by recursively traversing the abstract syntax tree of the function and noting the presence of arrays, addressable variables, and calls to alloca(). The algorithm also detects arrays that are nested arbitrarily deep in struct and union types.

The inst_pac pass inserts the inline assembly block with the code in Figures 3.1a and 3.2a as the first instruction into each protected subroutine. While such a simple solution is possible for the function prologue due to disabled -fshrink-wrap optimization, not all function epilogues are at the end of the subroutine and therefore require special attention. A challenge faced during development of the GCC plugin was identifying all the function epilogues generated by the compiler and placing the authentication code precisely after each one. Rather conveniently, the RTL representation at this stage of code generation provides:

1. (note ... NOTE_INSN_EPILOGUE_BEG) expressions marking beginning of each epilogue;
2. (barrier ...) expressions placed in the instruction stream where the control flow cannot advance further (e.g., after an unconditional jump or a return instruction);
3. (insn/f ...) with an /f flag indicating that the instruction is related to frame allocation code.

Therefore, each epilogue can be located by searching the RTL representation of the subroutine for notes marking their beginning (1). After that, its end is identified by iterating through the RTL expressions until a barrier (2) is encountered while keeping track of the most recently seen frame-related (3) instruction. If no frame-related instructions were encountered, the function does not allocate a stack frame and the epilogue was optimized out by the compiler. In this case, a stack buffer overflow is unlikely and no return address authentication code is inserted.

3.3 Linux Kernel Extension

There are multiple reasons for the pointer authentication service to be implemented as a component of the operating system. Firstly, this design choice improves the security of the technique by making the secret key inaccessible from the userspace. This is an advantage over other software-based pointer authentication techniques relying on information hiding using randomization, such as CCFI [26] and CPI [32]. Secondly, per-task management of the keys, installation of shared pages and support of multithreaded applications requires extending the OS scheduler and the virtual memory management subsystem.

Unfortunately, the changes introduced into the kernel subsystems are too intrusive for the extension to be implemented as a loadable kernel module. Instead, the extension constitutes a patch that is applied directly onto the source tree of the Linux kernel version 5.17. It introduces a net addition of 1307 lines of code across 27 files. The majority of source code changes affect the virtual memory subsystems of AArch64 and x86-64 and the core scheduler code.

3.3.1 Configuration options

Before the Linux kernel can be compiled, it is necessary to configure its source code tree. This is done using graphical or text-based programs shipped together with the kernel source code, such as `nconfig`, `menuconfig`, `xconfig`, and others. These programs manipulate the configuration database, which is organized into a tree structure and is described using the Kconfig language [44]. Due to the enormous size of its code base, the version 5.17 of the Linux kernel includes nearly 18 thousand of such configuration options. The kpac extension introduces several options into the Kconfig configuration system to customize static aspects of the pointer authentication service. These options are listed in Figure 3.5 and boil down to two aspects: the userspace mapping address, which defaults to `0x9AC00000000`, and the selection of the hashing algorithm. The pointer authentication mechanism can use different cryptographic algorithms to generate the pointer authentication codes. Within the scope of this work, support for two algorithms was developed and evaluated. These algorithms include *xxHash* [45] digest algorithm, which is designed to be fast but not cryptographically strong, and the original QARMA algorithm used by the ARMv8.3-A architecture with better security properties. Apart from that, the hashing backend can be disabled completely. This option causes the pointer authentication service to return the pointer without modifications and was used to evaluate the efficiency of the communication mechanism.

Before the kpac pointer authentication service can be used, it needs to be configured according to the use-case and the topology of the system it runs on. Following the commonly exercised practice of the Unix-based operating systems *“everything is a file”*, these runtime settings are

```
1 +- Security options
2    +- Enable software-emulated pointer authentication [bool]
3       +- Userspace mapping address                    [hex]
4       +- Pointer authentication backend               [choice]
5          +- None (DEBUG)
6          +- xxHash hashing algorithm
7          +- ARMv8.3 QARMA block cipher
```

Figure 3.5 – Build-time configuration options of the kpac extension.

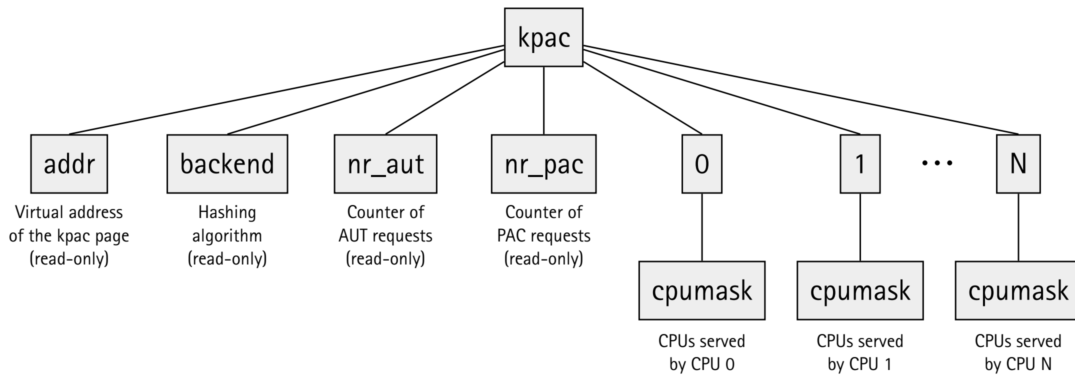


Figure 3.6 – Layout of the kpac tree in the debugfs filesystem.

exported in the *debugfs* filesystem of the Linux kernel. The debugfs filesystem is a pseudo file system that provides means for kernel developers to easily export internal data structures of the kernel subsystems into the userspace [46]. It is usually mounted under the `/sys/kernel/debug` directory. The kpac extension creates a directory named `kpac` in the debugfs file system and populates it with contents summarized in Figure 3.6. The contents include read-only information about the build configuration (virtual address, hashing algorithm) as well as information about the amount of processed requests. Apart from that, the kpac directory includes a subdirectory with a single `cpumask` file for each live CPU core of the system. These subdirectories represent the mapping of the CPU cores providing and receiving the pointer authentication service $U \to \mathscr{P}(U)$, where U is the set of online CPU cores. Writing a set of the CPU cores $M \in \mathscr{P}(U)$ into the `cpumask` file of the CPU core $N \in U$ causes the kernel to start the pointer authentication service on the CPU core N, such that it would serve the CPU cores in M. The set cannot include the CPU core that is serving it or intersect with the sets served by other CPU cores. If such violation is detected, the kpac extension rejects the input and prints an error message into the kernel log. The sets are represented using comma-separated ranges of CPU cores, such as “`0,2,4-6`”. This syntax is used repeatedly in the Linux kernel interfaces and is described extensively in the documentation [47]. Writing an empty set represented by an empty string into the `cpumask` file shuts down the service on the respective CPU core.

3.3.2 Multiprocessing support

One of the goals of this work was finding a solution for support of multithreaded applications. This also benefits the original PAC-PL [5] mechanism, which does not have such support. Nevertheless, it has a similar architecture, since the userspace applications communicate with the field-programmable gate array (FPGA) using memory-mapped I/O. Therefore, the principles introduced here may also be used to enable multithreading for PAC-PL.

Effectively, to support multiprocessing in general, it is sufficient to provide the pointer authentication service to multiple CPU cores concurrently. In case of multiple concurrently executing singlethreaded applications, such arrangement is trivial and could be implemented by installing the respective kpac page into a process whenever it is assigned or migrated to a particular CPU core. Unfortunately, this idea cannot be applied to multithreaded processes, since all the threads share a single virtual address space. In other words, installing the kpac page for one thread means it is immediately visible in all other threads. Scheduling these threads on

different CPU cores would lead to data races due to unsynchronized accesses to the kpac page. A naive solution would be to serialize these accesses by using synchronization primitives in the pointer authentication code. However, such solution introduces another level of synchronization in addition to communication with the pointer authentication service and would lead to a huge performance bottleneck.

There are multiple ways to decouple concurrent threads from each other when it comes to accesses to the kpac page. A solution that was considered in the early stages of the development involved using the *thread-local storage* (TLS) [48] mechanism. The TLS provides each thread with its own instances of *thread-local* variables defined in the global scope, despite the fact that the threads share a virtual address space. It is set up and managed by the system implementation of the standard C library and can use different mechanisms depending on the system or even the way the application was compiled and linked. This mechanism is implemented fully in the userspace and, hence, the OS kernel is not aware of its existence. Therefore, this idea was abandoned, since adding such functionality to the Linux kernel would be out of scope of this work.

The address space generations introduced by Rommel et al. [49] and implemented as a Linux kernel extension provide means to create multiple address spaces within a single process. These address spaces can be selected on per-thread basis and used to provide threads with different versions of some memory regions, while keeping the rest of the address space in sync. The address space generations could be extended to provide each thread with its own view of the kpac page. However, such solution was considered too heavy, since it involves duplicating the internal Linux structures representing the address spaces along with their respective page tables. This requires additional page-fault handling and leads to inconsiderate usage of the translation lookaside buffer (TLB), and thus potentially introduces a significant performance slowdown.

Therefore, the technique of *per-CPU PGDs* that has minimal performance impact and provides each CPU core with its own kpac page was developed. In this context, PGD (Page Global Directory) is a name that has established itself in the Linux kernel source code and refers to the top-level page table. Per-CPU in this context carries the meaning *"per CPU core"*, since the Linux kernel source code typically refers to CPU cores as simply CPUs. This technique is based on maintaining a separate version of each address space for each CPU core. Doing so does not involve duplicating the whole page table tree, but rather only its top level (PGD). In these

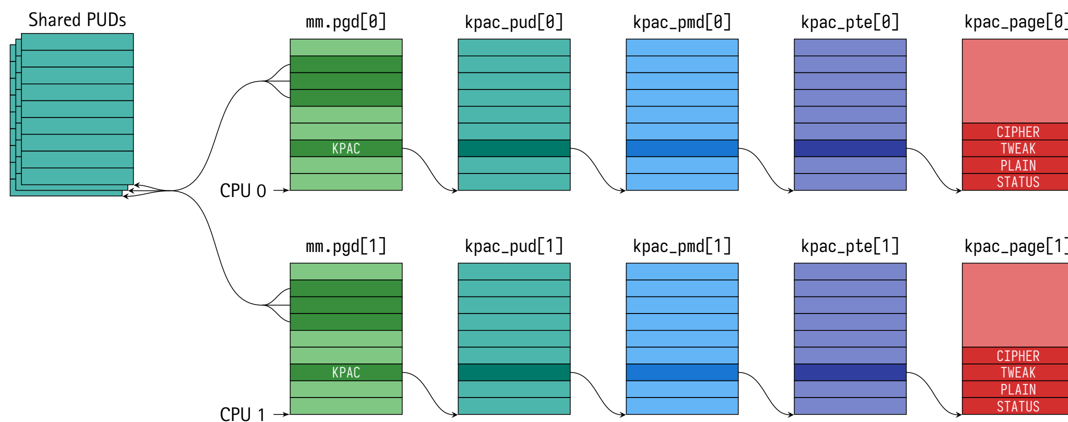


Figure 3.7 – Page-table arrangement introduced by per-CPU PGDs.

duplicated top-level page tables, all but one entry refer to the same underlying page tables (PUDs). Therefore, any modifications of these underlying page tables or in pages referenced by them are instantly seen by other CPU cores in their versions of the address space. Furthermore, any modifications done to entries in the top-level page tables are manually mirrored in the top-level page tables of other CPU cores.

The single entry that is not shared among its sibling versions of the top-level page table refers to the *kpac page table* of the respective CPU core. During the system initialization, the kpac extension allocates these kpac page tables along with the kpac page for each CPU core separately and chains them into a page-table subtree. When a new address space is created by means of `fork()` or `exec()` system calls, the extension inserts these page-table subtrees into the PGDs of the corresponding CPU cores. The Linux scheduler was modified to select the appropriate PGD according to the CPU core it runs on during a context switch. This arrangement, illustrated in Figure 3.7, ensures that each core of the system executing a userspace task sees its own kpac page. When applied to multithreaded applications, each concurrently running thread sees a different kpac page according to the CPU core it runs on, while the rest of the address space stays shared.

Assuming a system with 4 KiB pages and 4-level page tables, reserving a single entry of the top-level page table for kpac page means that $\frac{1}{512}$ or 512 GiB of the virtual address space cannot be used by the applications anymore. However, this is barely a concern considering the enormous size of the virtual address space (256 TiB). That being said, the technique was tested with 5-level paging [10] introduced in recent x86-64 processors by Intel. It is also compatible with the page-table isolation, or KAISER [50], a protection technique used by the Linux kernel to mitigate the notorious Meltdown [51] attack.

3.3.3 Kernel threads for pointer authentication

The pointer authentication service is implemented using *kernel threads*. The kernel threads are similar to regular userspace tasks and are generally treated same by the OS scheduler, but execute entirely in the kernel space. To avoid superfluous context switching, they do not possess an address space of their own and reuse the address space of the previous process. The kernel threads are used to delegate critical system-management tasks, such as flushing disk caches, swapping out unused pages, or servicing network connections [52, p. 123].

The kernel thread `kpacd/n`, where `n` is the ID of the CPU core it runs on, is started as soon as non-empty set of CPU cores is stored in the respective `cpumask` file. Although the priority of the kpacd thread is automatically set to the maximum, it is recommended that it is started on a CPU core that is completely isolated from other tasks in the system. On Linux, this can be achieved by using the *isolcpus* command-line parameter [47] or the *cpuset* mechanism [53]. The kernel thread polls one or several kpac pages according to its `cpumask` list and fulfills requests which are submitted by the user applications via these pages.

To synchronize with the kernel scheduler, the kpacd threads maintain a mask of the globally polled CPU cores protected with a spin lock. In contrast to a simple union of the `cpumask` sets in the debugfs, this mask does not include kpac threads that are temporarily scheduled away. To achieve this, the kpacd threads do not allow preemption and instead call the scheduler manually once their time slice expires, adjusting the mask of polled CPU cores beforehand. The following section describes this synchronization mechanism in depth.

3.3.4 Modifications to the OS scheduler

To provide management of the keys and enforce isolation between different processes scheduled consecutively on the same CPU core, the pointer authentication service operates jointly with the kernel scheduler. Furthermore, the *thread control block* (TCB), which is an internal data structure representing the state of the tasks, is extended to hold the *kpac context*. The kpac context includes the key used for computation of cryptographic signatures as well as the contents of the kpac page registers at the moment the task was last scheduled away. To manage this state, the extension introduces two new callbacks into the Linux kernel scheduler, namely `kpac_finish()` and `kpac_switch()`.

The `kpac_finish()` callback is executed early in the scheduling process, and the next task chosen by the scheduler is not yet known. This callback resolves any pointer authentication requests that were made by the task, but not yet finished at the moment the task was preempted. It starts by verifying the value of the `STATUS` register in the kpac page. If it contains zero, then no pending requests were interrupted by the preemption and the callback returns, resuming the scheduling process. Otherwise, the callback locks the mask of the globally polled CPU cores and checks whether the current CPU core is contained in this mask. At this point, there are two possible cases:

- The CPU core is contained in the mask and there is a concurrent kpacd thread about to fulfill this request. In this case, the callback spins until the value of the `STATUS` register changes to zero, releases the lock, and returns.
- The CPU core is not contained in the mask of the polled CPU cores. This means that this CPU core is either not assigned to any kpacd thread, or the respective kpacd thread was temporarily scheduled away. In this case, the callback fulfills the request on the spot, releases the lock, and returns.

For this mechanism to work, following requirements have to be satisfied:

- To avoid deadlocks, the kpacd threads cannot be scheduled away if there is a waiter in the `kpac_finish()` callback. This condition can be easily overapproximated by checking whether the mask of the polled CPU cores is locked. Once rescheduling is requested by the kernel, the kpacd thread *attempts* to lock the mask, since it is also required to adjust it before yielding the CPU core. If the attempt is unsuccessful, i.e., the lock is taken, no rescheduling is done in favor of one more polling pass to unblock any potential waiters in the `kpac_finish()` callback.
- To avoid race conditions, the kpacd threads cannot resume polling after being scheduled in if another CPU core is fulfilling the request concurrently in the `kpac_finish()` callback. This requirement matches the requirement to adjust the mask before polling is resumed. Since doing so implies taking the lock, the kpacd thread will spin, waiting for the concurrent `kpac_finish()` call to finish and release it.

The `kpac_switch()` callback is executed immediately prior to the context switch when the next task is already determined. This callback does nothing more than saving the kpac context of the previous task and restoring the kpac context for the next task. Managing the contents of the kpac page in such way ensures separation between the processes and allows continuing the execution at the same point if the task was preempted during request preparation. The callback also adjusts the global reference to the task running on the respective CPU core. This reference is used by the kpacd threads to match the ID of CPU core to the key of the process in the TCB. The key is generated in the `kpac_exec()` callback that is inserted into the final stage of the `exec()` system call.

3.4 Summary

This chapter introduced the architecture of the CFI solution, which consists of the Linux kernel extension and a GCC plugin. During development of this technique, a focus was laid on minimizing the performance overhead, while keeping the bar of security guarantees high. Particularly close attention was paid to aspects that would benefit other works, such as PAC-PL [5]. These aspects include support of compiler optimizations for the compiler plugin as well as support of multithreaded applications for the Linux kernel extension. The pointer authentication service is provided by kernel threads that fulfill pointer authentication requests submitted by other CPU cores via a shared page in memory. This service is integrated tightly into the system scheduler to ensure separation between pointer authentication transactions in different tasks and keep the communication protocol simple without sacrificing performance. To support multiprocessing, this work proposes a lightweight technique of per-CPU PGDs, which involves maintaining separate versions of address spaces for each CPU core that differ only in the top-level page table.

4 EVALUATION

This chapter presents the evaluation of the Control-Flow Integrity (CFI) solution for return address protection developed within the scope of this thesis. It starts with Section 4.1 analyzing the effectiveness of multiple protection strategies implemented in the PAC-SW plugin based on the past security vulnerabilities in the GNU C library. Then, the influence of various hashing algorithms and protection scopes on the performance overhead are assessed in Section 4.2 using a collection of synthetic benchmarks. Finally, Section 4.3 evaluates the performance overhead, multicore scalability, and energy efficiency of the technique in a real-world multithreaded environment.

4.1 Protection Effectiveness

The *Common Vulnerabilities and Exposures* (CVE) is an effort launched in September 1999 and operated by The Mitre Corporation together with the NIST (US National Institute of Standards and Technology) to maintain a database of publicly disclosed security vulnerabilities in software projects [54]. In September 2022, it contained more than 180 thousand CVE records that are scored in severity based on several metrics, including impact, exploitability and the involved attack vectors. As this study was conducted in July 2022, the database contained 131 (CVE-1999-0199 through CVE-2022-23219) records for the security vulnerabilities involving the GNU C library project. The GNU C library is the standard and most widespread implementation of the standard C library on GNU/Linux systems. To compare the effectiveness of various protection scopes, the vulnerabilities in the CVE database involving the GNU C library were qualitatively assessed in three aspects:

- Does the security vulnerability involve a stack-based buffer overflow? If yes, in which functions?
- What is the minimal protection scope of the PAC-SW plugin to include the vulnerable functions?
- Some vulnerabilities involve overflowing a buffer received from a function that links with and calls into the GNU C library. Assuming this buffer is allocated on the stack of the calling function, what would be the required protection scope to protect its return address?

To answer the last question, the typical use case of the function involving the vulnerability and the type of the buffer are analyzed. For example, if the vulnerability involves overflowing an `int` buffer, `array` protection scope is assumed, since it is the smallest scope that would include a hypothetical function with an `int` array on the stack.

CVE ID	Location of the buffer overflow	Required protection scope
CVE-2022-23219	glibc function `clnt_create()`	`char`
CVE-2022-23218	glibc function `svcunix_create()`	`char`
CVE-2020-29573	caller of `printf()`	`char`
CVE-2020-10029	glibc function `__ieee754_rem_pio2l()`	`array`
CVE-2020-6096	caller of `memcpy()`	`strong`
CVE-2020-1751	caller of `backtrace()`	`array`
CVE-2018-1000001	caller of `getcwd()`	`char`
CVE-2018-11236	glibc function `__realpath()`	`char`
CVE-2016-1234	glibc function `glob()`	`char`
CVE-2015-8982	glibc function `strxfrm()`	`char`
CVE-2015-8779	glibc function `catopen()`	`char`
CVE-2015-1781	caller of `gethostbyname_r()`	`char`
CVE-2015-1473	glibc function `_IO_vfscanf_internal()`	`char`
CVE-2015-1472	glibc function `_IO_vfscanf_internal()`	`char`
CVE-2014-9761	glibc function `__nan()`	`char`
CVE-2013-4237	caller of `readdir_r()`	`char`
CVE-2012-4424	glibc function `strcoll()`	`char`
CVE-2012-4412	glibc function `strcoll()`	`char`
CVE-2012-3480	caller of `strtod()`	`char`
CVE-2012-3405	glibc function `vfprintf()`	`char`
CVE-2012-3404	glibc function `vfprintf()`	`char`
CVE-2012-0864	glibc function `vfprintf()`	`char`

Table 4.1 – Security vulnerabilities discovered in the GNU C library that involve a stack-based buffer overflow.

Table 4.1 demonstrates the results of the study. Out of 131 analyzed security vulnerabilities, 22 contained vulnerabilities that could be used to cause a stack-based buffer overflow. More than two thirds of these involve a buffer overflow within the GNU C library itself. In all but one of these vulnerabilities, it would be enough to use the `char` protection scope to protect the return address of the vulnerable function and prevent control-flow redirection. However, the reason the vulnerable functions are included in the `char` protection scope is not necessarily a character buffer within the function. The GNU C library extensively uses the `alloca()` compiler builtin, which allocates additional space on the function's stack frame at runtime. Since the `alloca()` builtin can be used to allocate an array of any type (including `char`), the protection scope `char` includes all functions that call `alloca()`. Besides, the `alloca()` builtin requires special caution and is often misused. In fact, several of the analyzed vulnerabilities involved incorrect usage of `alloca()` that caused a memory corruption on the stack. The only vulnerability that is not covered by the `char` protection scope and requires the protection scope of at least `array` is CVE-2020-10029. This vulnerability involves overflowing a three-element `double` buffer in the floating point range reduction code of the GNU C library.

Seven of the analyzed vulnerabilities enable the attacker to overflow a buffer supplied by the caller. The majority of them involve standard library functions that typically receive a character buffer from the user, such as `printf()`, `getcwd()`, `strtod()`, and others. Thus, assuming the passed buffer is located on the stack, these vulnerabilities require the protection scope `char` to prevent their exploitation. The remaining two vulnerabilities require a larger protection scope,

namely `array` for CVE-2020-1751 and `strong` for CVE-2020-6096. While the former vulnerability involves an array of `void` pointers, the latter resulted from a memory corruption bug in the `memcpy()` function. Since it is also customary to pass the address of local variables to `memcpy()`, the required protection scope for CVE-2020-6096 is assumed to be `strong`.

Discussion

Despite the fact that the `char` protection score covers 86 % of the software vulnerabilities discovered in the GNU C library since 1999, a big portion of the vulnerable functions include a call to `alloca()`. Regardless whether the security vulnerability is caused by incorrect usage of `alloca()` or not, these functions are included in the protection scope `char`. Since such extensive usage of the `alloca()` builtin is not typically seen in other projects, I consider these results to be somewhat misleading. Nevertheless, improper string manipulation is still a very common source of security vulnerabilities in C projects [2]. This makes the `char` protection scope an attractive choice when low performance overhead is desired.

No vulnerabilities assessed in this study required a protection scope larger than `strong` to prevent their exploitation. Functions not included in the `strong` scope do not expose references pointing to their stack to the outside world and are fully self-contained from the attacker's point of view. Thus, a stack-based buffer overflow in such functions is extremely unlikely, making the protection scope `all` an overkill in most cases. To provide a better context for these results and examine the nature of the trade-off between security and performance, the following section focuses, inter alia, on the performance footprint of various protection scopes.

4.2 Synthetic Benchmarks

The influence of the protection scope and the hashing algorithm on the performance overhead is evaluated using the TacleBench [55] benchmark suite. TacleBench is a collection of 57 synthetic benchmarks from different vendors, which are fully self-contained (do not link with any libraries), platform-independent and represent a CPU-intensive workload. For this evaluation, the benchmarks were compiled without optimizations and performed on different systems for each architecture. The x86-64 machine used for these measurements featured a quad-core Intel Core i5-6500T @ 2.50 GHz CPU and 8 GiB of memory. The AArch64 machine was a Raspberry Pi 4 single-board computer with 8 GiB of RAM and a quad-core Cortex-A72 processor running at 1.8 GHz. For the operating system, both machines ran a Debian GNU/Linux 11 system including the Linux kernel 5.17 with the kpac extension. The x86-64 system had the kernel page-table isolation mechanism [50] enabled. To reduce the operating system noise, the benchmarks and the pointer authentication service were pinned on separate CPU cores with fixed frequency. These CPU cores were isolated from the rest of the system and configured to run in tickless mode [56] using the *isolcpus* [47] Linux kernel command-line parameter.

4.2.1 Impact of the hashing algorithms

Figure 4.1 illustrates the distribution of the performance overhead in terms of authentication requests performed per benchmark run. The benchmarks were compiled with the protection scope `all`. Since the TacleBench benchmarks are fully deterministic and perform equal amount of authentications per each run, each benchmark is mapped to a value on the x-axis. Thus, for each point on the scatter plot, the corresponding runs with different hashing algorithms can

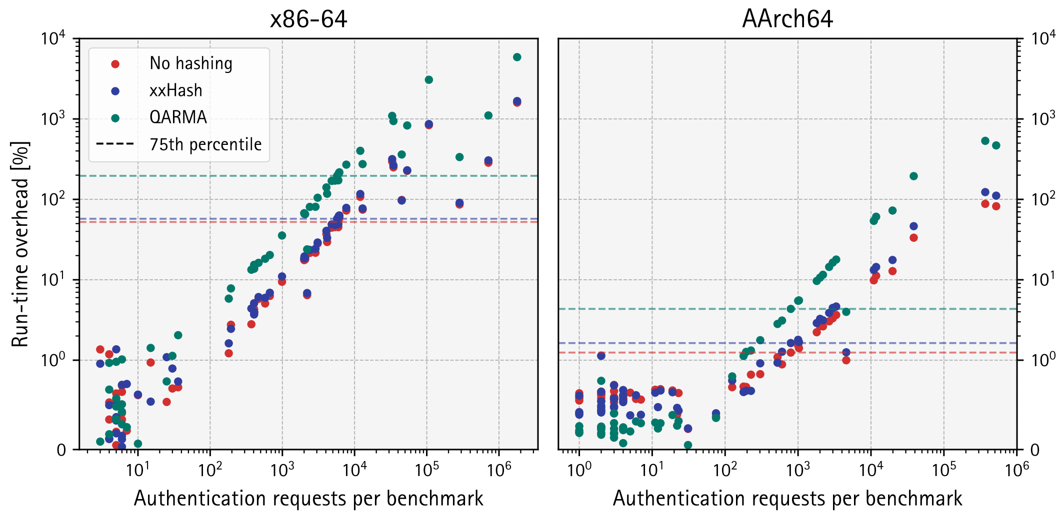


Figure 4.1 – TacleBench: Distribution of the performance overhead in benchmarks featuring different hashing backends. The benchmarks were compiled with the protection scope `all`.

Architecture	Hashing backend	Third quartile [%]	Overhead factor (σ)
x86-64	QARMA	196.15	3.70 (0.203)
	xxHash	57.38	1.08 (0.04)
	None	52.35	1
AArch64	QARMA	4.34	4.93 (0.792)
	xxHash	1.63	1.29 (0.077)
	None	1.24	1

Table 4.2 – TacleBench: Exact values of the third quartile in Figure 4.1 and the average overhead factor for different hashing backends.

be found on the same axis. The results show a broad distribution of performance overhead for various benchmarks, ranging from near 0 to 5872 % overhead on an x86-64 system with QARMA hashing backend. The AArch64 system with QARMA hashing performs better with an overhead of up to 535 %, which is an order of magnitude lower than x86-64 for the worst-performing benchmarks. In case the xxHash hashing algorithm is used, the performance overhead can be as high as 1668 % for x86-64 and 122 % for AArch64 systems. With no hashing, the worst-case performance overhead is 1594 % for an x86-64 system and 88.6 % for an AArch64 system.

While the numbers mentioned in the previous paragraph are too large for any practical application, they represent a pathological case. As indicated by the dashed lines in Figure 4.1 representing the third quartile (75th percentile) of the overhead distribution, the majority of the benchmarks exhibit significantly lower performance overhead. For instance, three quarters of the benchmarks in TacleBench demonstrate an overhead that is lower than 57.38 % on x86-64 and 1.63 % on AArch64 for xxHash. For the QARMA hashing backend this figure is somewhat higher and amounts to 196.15 % on x86-64 and 4.34 % on AArch64. The exact values of the third quartiles are listed in Table 4.2.

A remarkable observation can be made by inspecting the relation between the performance overheads of various hashing backends for each benchmark: the difference between the performance overheads of different hashing algorithms is a constant factor. The values of this factor computed by averaging the increase in performance overhead with respect to the run with no hashing are listed in Table 4.2, together with their standard deviation. To avoid benchmarks with low overhead and high relative error distorting the value, the computation only included the benchmarks with an overhead of at least 5 %. Particularly low standard deviation proves that this heuristic can be used to estimate performance overhead for various hashing algorithms.

Discussion

The hashing algorithm has a very significant influence on the performance overhead of the pointer authentication mechanism. It can be observed from the measurements that the relation between different hashing algorithms can be reduced to a single constant factor. The QARMA algorithm, while designed specifically for pointer authentication, cannot be implemented as efficiently in software and increases the performance overhead by roughly a fourfold. In contrast to QARMA, the xxHash algorithm comes with much lower cost that is comparable to the run without hashing. The performance overhead of hashing can be reduced even further, while maintaining high security standard, by utilizing the hardware-accelerated cryptography extensions present in modern CPUs. Examples of such extensions are AES-NI [57] instructions by Intel and the optional cryptographic extensions implementing the AES and SHA256 algorithms in the ARMv8 architecture [9].

While the performance overhead of the individual benchmarks in the TacleBench benchmark suite ranges from very low to extremely high values, considering the benchmark suite as a whole yields a more informative picture. Apart from several outliers, the majority of benchmarks show a relatively low performance overhead even when using the protection scope `all`. In the next section, the mitigations of the performance overhead are analyzed further by comparing the difference in performance overhead between various protection scopes.

4.2.2 Impact of the protection scopes

Figure 4.2 illustrates the differences between various protection scopes in the 12 benchmarks with the largest performance overhead for each architecture. The upper plot shows the amount of protected functions, while the lower plot demonstrates the performance overhead. It is noteworthy that the protection scope `all` does not protect 100 % of the functions on AArch64, in contrast to x86-64. This is due to omission of the leaf functions, which do not save their return address on the stack on AArch64 and thus require no protection from stack-based buffer overflows.

On x86-64, the overhead in case the protection scope `char` is used does not exceed 100 %, while for 9 out of 12 benchmarks, the overhead lies below 2 %. The protection scope `array` increases this figure by about a tenfold for 4 benchmarks and by up to a threefold for another 6 benchmarks. Rather surprisingly, the `array` protection does not influence the performance overhead of the `ammunition` and `anagram` benchmarks. The reason for this is that these benchmarks are based on string manipulation and contain character arrays in functions that are frequently called. Thus, the protection scope `char` already covers all the "hot" paths in the code, and `array` does not introduce a significant impact. While the protection scope `strong` triples the overhead for three benchmarks, it does not introduce a substantial change in the overall picture of the performance overhead. For 11 out of 12 analyzed benchmarks, the performance overhead stays

below 100 %, which is a reasonable figure for practical application of the solution. The protection scope `all`, in contrast, increases the performance overhead by at least another tenfold for each benchmark in the analyzed set.

A similar tendency, but of a different magnitude, can be observed on AArch64. With exception of `ammunition`, the performance overhead of protection scopes `char`, `array`, and `strong` stays below 10 % for 11 out of 12 benchmarks. For the `ammunition` benchmark, the performance overhead is 30 % for `char` and 48 % for both `array` and `strong` protection scopes. While protecting all functions raises the performance overhead considerably, it still stays below 20 % for the majority of benchmarks. For the worst-performing benchmark `ammunition` this figure amounts to 90 %.

Discussion

The selection of an appropriate protection scope allows to mitigate the performance overhead further, bringing even the worst-performing benchmarks into a range that is acceptable in practical applications. In this way, choosing the proper protection scope constitutes a trade-off between degree of security and run-time performance.

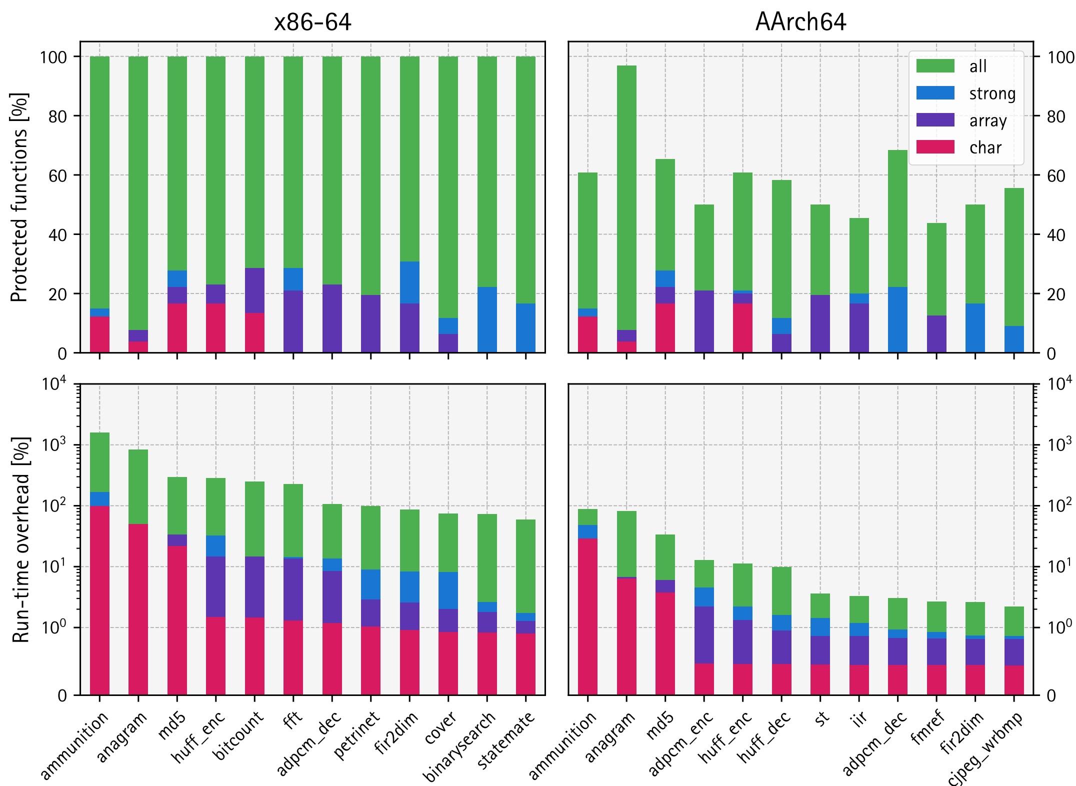


Figure 4.2 – TacleBench: Impact of various protection scopes on the benchmarks with high performance overhead. The contribution of each protection scope with respect to its closest subset is represented by the respective segments of the bars. The pointer authentication service performed no hashing for this experiment.

The analysis shows that as long as the protection scope stays below `all`, the overhead remains in the reasonable range. In agreement with the conclusion made in Section 4.1, protecting all functions is an excessive measure that does not introduce much value in terms of security. Therefore, the protection scope `strong` represents the most adequate compromise between the performance overhead and security in all benchmarks analyzed in this section.

4.3 Multithreaded Application

The previous sections analyzed the influence of various variables on the performance of synthetic benchmarks. These synthetic benchmarks do not necessarily represent a realistic usage of the CFI solution presented here. In this section, the solution is evaluated in its practical aspects by integrating it into a real-world multithreaded application. For this purpose, the distributed memory object caching system *memcached* [58] was selected. This choice is motivated by memcached's simple code base written in plain C, the availability of benchmarking tools and its frequent use on commercial servers.

4.3.1 Run-time performance and multicore scalability

To examine the multicore scalability, the performance is evaluated on high-performance servers with numerous CPU cores. For x86-64, a PowerEdge R740 server with two 24-core Intel Xeon Gold 6252 CPUs @ 2.10 GHz and 374 GiB of RAM was used. The hyperthreading and frequency scaling were disabled to avoid distorting the measurements. Circumventing the costly communication between NUMA (Non-Uniform Memory Access) nodes, both the 12 memcached server threads and up to 12 pointer authentication services were pinned on cores within a single NUMA node. The benchmarking client *memtier* [59], executed on the same machine but on a separate NUMA node, ran with 12 concurrent threads as well. The AArch64 measurements were carried

		GET request latency [ms]				
		Average	P50	P99	P99.5	P99.9
x86-64	`nil-0`	2.032	1.876	2.990	3.420	10.101
	`strong-1`	2.087	1.714	8.438	17.558	21.790
	`all-12`	2.930	2.788	5.812	6.778	12.262
	`all-6`	3.296	3.108	6.602	7.354	13.259
	`all-2`	4.445	4.290	8.815	9.428	14.902
	`all-1`	7.599	7.515	13.081	14.924	19.088
AArch64	`nil-0`	0.472	0.459	0.704	0.715	0.828
	`strong-1`	0.615	0.591	1.201	1.222	1.569
	`all-12`	1.503	1.484	1.941	2.777	4.620
	`all-6`	1.541	1.521	1.866	2.898	4.655
	`all-2`	2.356	2.340	2.719	3.349	5.877
	`all-1`	4.663	4.625	5.849	6.646	8.375

Table 4.3 – Memcached: Latency spectrum for various protection scopes and kpac configurations. The number next to the protection scope stands for the amount of assigned kpac instances.

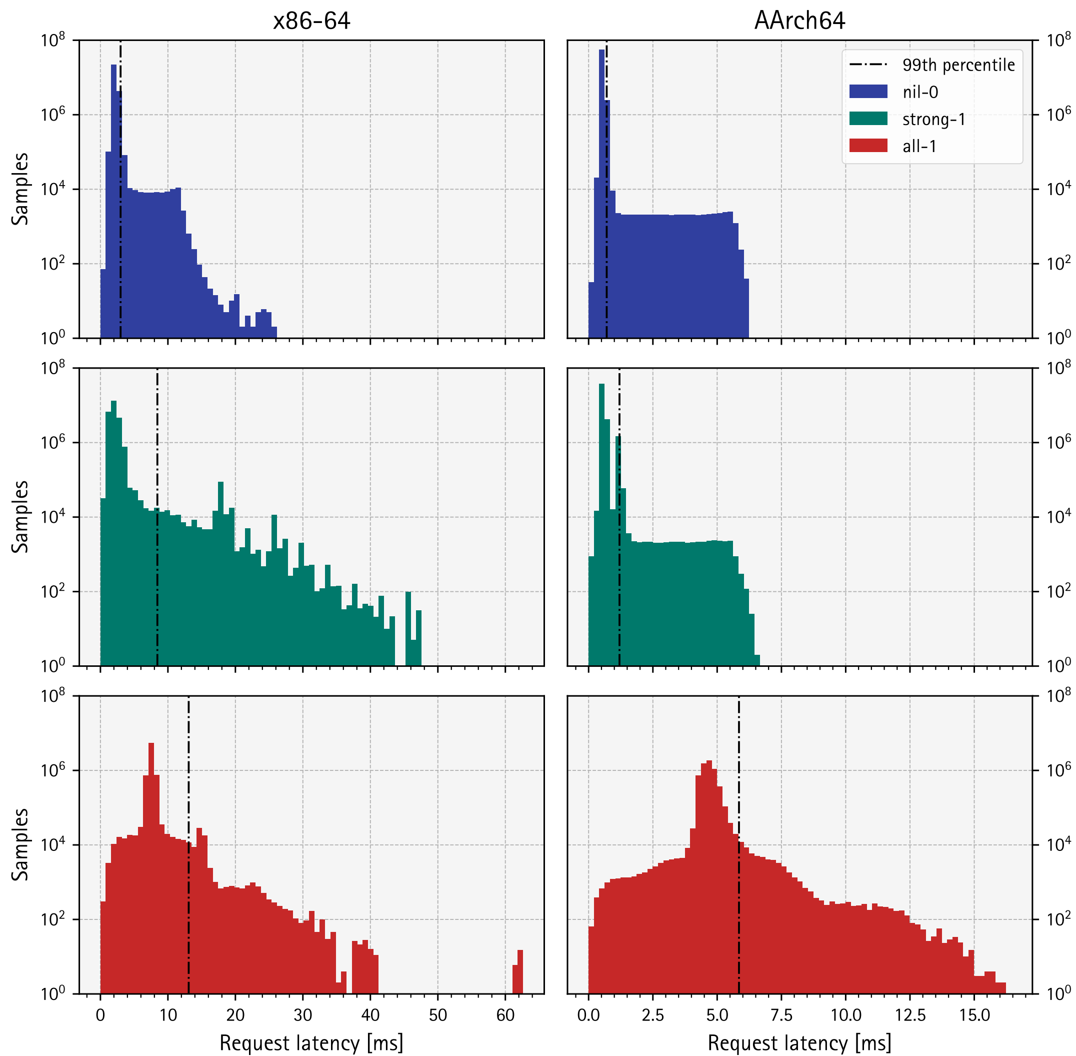


Figure 4.3 – Memcached: Distribution of the request latencies for various protection scopes and a single kpac instance.

out on a Gigabyte R152-P30 server featuring an Altra Q80-30 CPU with 80 cores clocked at 3 GHz and 250 GiB of RAM. The AArch64 arrangement corresponded to the x86-64 experiment, i.e, 12 CPU cores assigned to each one of memcached, memtier, and kpac. On the software side, both machines ran a Debian GNU/Linux 11 system on Linux kernel 5.17 with kpac extension featuring the xxHash hashing backend.

The memtier benchmark used for these measurements was modified to output individual latencies of GET requests handled by the memcached server. Figure 4.3 demonstrates the histogram of latency distribution for three protection scopes and a single pointer authentication service. The latency spectrum for various configurations is summarized in Table 4.3. The number next to the protection scope stands for the amount of assigned pointer authentication services.

For instance, all-6 is used to denote the protection scope all with six pointer authentication threads that are assigned to the CPU cores running memcached.

The change in average response latency for strong protection scope computes to 2.7 % for x86-64 and to 30.3 % for AArch64. In general, the pointer authentication has more pronounced impact on the latency distribution then on the average value by broadening the spectrum. For instance, the increase of the 99th percentile latency is 182.2 % on x86-64 and 70.6 % on AArch64 for the protection scope strong. As can be observed in the latency histogram, the protection scope strong introduces a second peak in the latency distribution on the AArch64 system, which also explains the higher average response latency.

Figure 4.4 shows a change in throughput for different protection scopes, which is computed as an amount of requests performed in a unit of time. The protection scope all with a single pointer authentication service reduces the throughput to 27 % on x86-64 and to 10 % on AArch64. The reason for this, apart from the increased amount of authentications, is the contention caused by a single pointer authentication thread serving too many cores at once. To reduce the load on this kpac thread, a second instance of the pointer authentication service can be introduced. Doing so doubles the throughput, raising it to about 45 % on x86-64 and 20 % on AArch64. According to Table 4.3, this increase in throughput corresponds to a reduction in average response latency of 41.5 % on x86-64 and 49.7 % on AArch64. By increasing the amount of pointer authentication threads to 6, the throughput is improved further to 62 % on x86-64 and 31 % on AArch64. At this point, the pointer authentication service is no longer saturated on AArch64, since additional increase of pointer authentication threads to 12 instances does not yield any measurable improvement in the request latency. On the other hand, configuring 12 pointer authentication threads on x86-64 does bring a modest benefit of boosting the throughput to 69 %.

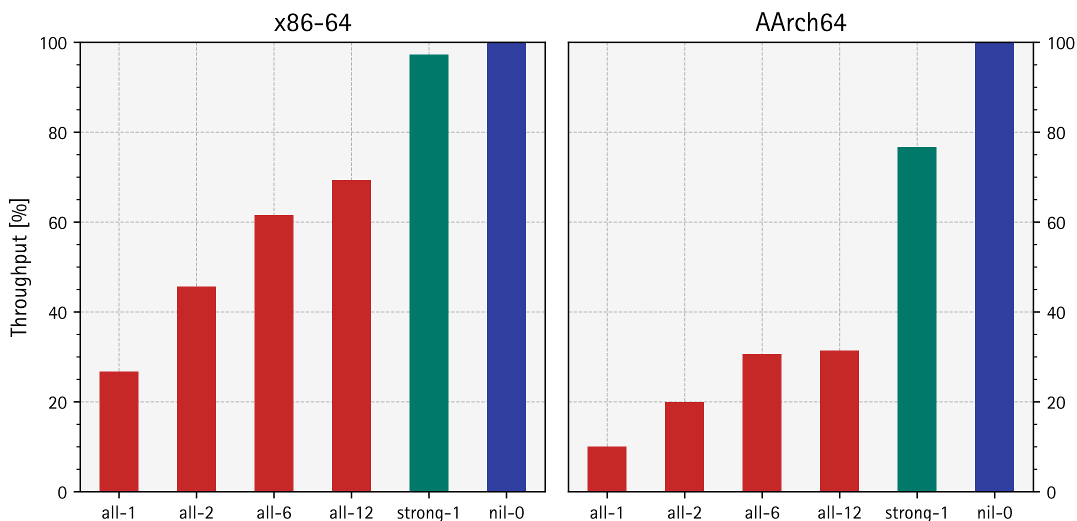


Figure 4.4 – Memcached: Throughput with respect to the baseline for various protection scopes and kpac configurations.

Discussion

Contrary to synthetic benchmarks, the pointer authentication mechanism performs better when applied to a multithreaded memory object caching system. A possible reason for this is that the additional runtime overhead introduced by return address protection is small compared to the time spent waiting or performing I/O in the OS kernel. Such workload profile is inherent to real-world applications, which are not purely CPU-bound, and represents the real performance cost of the return address protection by pointer authentication more precisely.

The `strong` protection scope has proved to be a reasonable compromise between the security improvement and the performance slowdown. However, if the applications running on the system produce more requests than the pointer authentication service can efficiently handle, the duration of the spin-wait loop increases drastically. This can be avoided by increasing the amount of pointer authentication services and spreading the pointer authentication load over multiple CPU cores.

4.3.2 Energy efficiency

To analyze the additional energy consumption introduced by the pointer authentication mechanism, a fixed workload is executed on the low-power systems introduced in Section 4.2. The workload consists of a memcached server that receives and handles $64 \cdot 10^6$ requests sent by another machine over network. Since only four CPU cores are available, the memcached server is configured to run three threads, which are served by a single kpac instance, if the pointer authentication is used. The pointer authentication backend used in this experiment is xxHash.

Figure 4.5 shows the total energy and the average power consumption of the workload measured on these machines. The return address protection on the x86-64 machine increases the average power from 34.3 W to 37.3 W, which corresponds to an increase of 8.7 %. On the AArch64 machine, the power increases by 16.2 % or from 5.86 W to 6.81 W. Despite the considerably lower power consumption of the AArch64 machine, it requires much longer time to process the workload. Thus, the total energy consumption of the AArch64 system is comparable to the x86-64 case. The energy consumed by the x86-64 system amounts to 5.3 Wh without pointer authentication and 6.6 Wh with pointer authentication. The figures of the AArch64

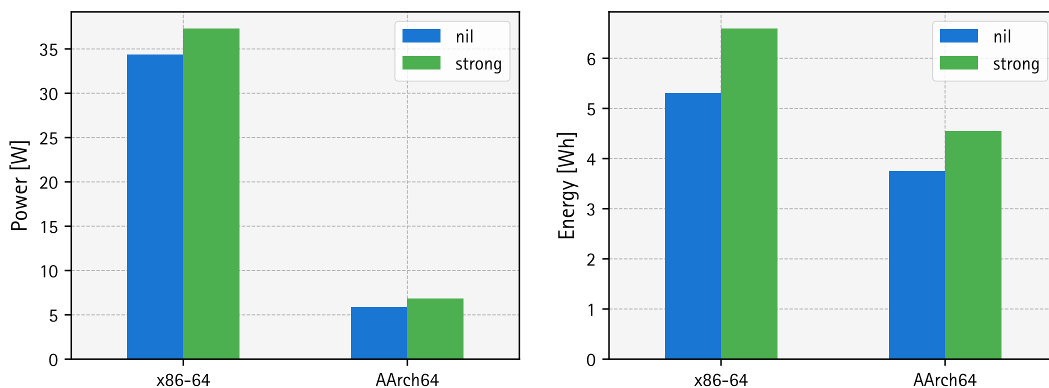


Figure 4.5 – Memcached: Energy consumption of the system for a fixed workload with and without pointer authentication. The workload consists of a memcached server that handles $64 \cdot 10^6$ requests sent over network by another machine.

system are somewhat lower and amount to 3.74 Wh without pointer authentication and 4.55 Wh with pointer authentication. On both systems, the increase in the total energy consumption does not exceed 25 %.

Discussion

Due to both the pointer authentication services and user application using busy-wait loops to achieve synchronization, the return address protection comes with an additional energy cost. The technique of busy waiting, while considered an anti-pattern in programming [60], is used here to avoid unnecessary context switching and achieve low performance overhead. Assuming a loaded system, the power consumption increase of up to 16.2 % might be considered an adequate price in cases where better security is desired and suitable hardware is not available.

4.4 Summary

To summarize, the pointer authentication mechanism and its usage as a return address protection technique are connected with an additional performance overhead. This overhead depends on several variables, such as the hashing algorithm, the protection scope, the amount of pointer authentication threads, and the profile of the workload. Analyzing this overhead using a set of synthetic benchmarks allowed to determine these dependencies and get a feel for the nature of the trade-off between security and performance. According to these results, the further evaluation focused on the return address protection using the xxHash hashing algorithm and the protection scope `strong`.

This configuration, introduced into the distributed memory object caching system *memcached*, demonstrated an increase in average response latency of 2.7 % on a x86-64 system and 30.3 % on an AArch64 system. Furthermore, the energy assessment showed a moderate increase in power consumption of 8.7 % on an x86-64 machine and 16.2 % on an AArch64 machine. Apart from that, it was observed that high amounts of authentication requests from multiple CPU cores can saturate a single pointer authentication service. To avoid this, the amount of requests handled per instance of the pointer authentication service should be held low by providing multiple kpac instances in multiprocessing environments.

5 CONCLUSION

This thesis presents a complete Control-Flow Integrity (CFI) solution for return address protection based on the technique of storing cryptographic signatures along with the return addresses. This technique has strong security guarantees that are achieved by computing the cryptographic signature in the OS kernel and not storing the secret key in the address space of the protected application. To achieve this, the Linux kernel was extended with a flexible pointer authentication service that imitates the ARMv8.3-A pointer authentication mechanism. The extension is tightly integrated into the process scheduler and virtual memory subsystem, allowing it to support multithreaded applications.

At the price of giving up CPU cores to the pointer authentication needs, this solution manages to avoid system calls and the high performance overhead associated with them by using a shared memory page to facilitate communication between the operating system and the userspace applications. As a part of this work, a GNU Compiler Collection (GCC) plugin to perform code instrumentation and automatically add return address protection to functions in userspace programs was developed. To mitigate the performance overhead further, the GCC plugin includes multiple heuristics that restrict protection only to functions that are deemed vulnerable.

An evaluation based on a collection of synthetic benchmarks and a multithreaded server application demonstrates that moderate performance overhead can be achieved without considerably sacrificing security. To elaborate, a performance overhead of 2.7 % on x86-64 and 30.3 % on AArch64 is observed when applying return address protection based on the hashing algorithm *xxHash* to functions that contain arrays or variables that had their address taken. For the same configuration, low-power consumer systems of both architectures exhibit a power consumption increase of up to 16.2 %. Such figures are acceptable in practical applications.

In the future, the software-emulated pointer authentication service can be extended to provide more operations and multiple keys per application, akin to the ARMv8.3-A mechanism. Apart from that, the list of hashing algorithms supported by the kernel extension can be expanded to include the hardware-accelerated cryptographic extensions commonly present on modern CPUs. Perhaps, the scope of software-emulated pointer authentication mechanism can be to extended onto the operating system itself by leveraging the system hypervisor. Furthermore, a comparison of the performance overheads between the software-emulated solution presented here and the FPGA-based solution of PAC-PL [5] on the same machine would allow to put the economic viability of both approaches in a context.

LIST OF ACRONYMS

ABI application binary interface
ASLR address-space layout randomization
CFG control-flow graph
CFI Control-Flow Integrity
CRA code-reuse attack
CVE Common Vulnerabilities and Exposures
FPGA field-programmable gate array
GCC GNU Compiler Collection
JIT Just-In-Time
LR link register
MAC message authentication code
MMU memory management unit
PAC pointer authentication code
ROP return-oriented programming
RTL register transfer language
SMP symmetric multiprocessing
SP stack pointer
SSP stack-smashing protection
TCB thread control block
TLB translation lookaside buffer
TLS thread-local storage
W⊕ X Write xor eXecute

LIST OF FIGURES

2.1 Virtual address and the corresponding 4-level page table walk 3
2.2 Implementation of an atomic counter incrementation on different architectures 6
2.3 Subroutine containing a stack-based buffer overflow vulnerability 7
2.4 Stack frame protected with a stack canary 8
2.5 Functional principle of the PAC instructions 10
2.6 Functional principle of the AUT instructions 11
2.7 Return address protection using the ARM pointer authentication 11

3.1 Additional code generated for return address protection on AArch64 19
3.2 Additional code generated for return address protection on x86-64 20
3.3 Exemplary usage of the `pac_scope` attribute 22
3.4 The RTL representation of an inline assembly block 23
3.5 Build-time configuration options of the kpac extension 24
3.6 Layout of the kpac tree in the debugfs filesystem 25
3.7 Page-table arrangement introduced by per-CPU PGDs 26

4.1 TacleBench: Performance overhead in benchmarks with different hashing backends 34
4.2 TacleBench: Impact of protection scopes on benchmarks with high performance overhead 36
4.3 Memcached: Distribution of the request latencies for various protection scopes . 38
4.4 Memcached: Throughput when using various protection scopes and kpac configurations 39
4.5 Memcached: Energy consumption of a fixed workload with and without pointer authentication 40

LIST OF TABLES

2.1 Allowed memory access reordering on AArch64 and x86-64 5

3.1 Registers in the kpac page and their usage during kpac requests 18
3.2 Protection scopes implemented by the PAC-SW plugin 21

4.1 Past GNU C library vulnerabilities that involve a stack-based buffer overflow . . 32
4.2 TacleBench: Values of the third quartile and the overhead factor for different hashing backends 34
4.3 Memcached: Latency spectrum for various protection scopes and kpac configurations 37

LIST OF LISTINGS

2.4 Return address protection using the ARM pointer authentication 11

3.5 Exemplary usage of the `pac_scope` attribute 22
3.6 The RTL representation of an inline assembly block 23
3.7 Build-time configuration options of the kpac extension 24

REFERENCES

[1] National Institute of Standards and Technology. *Full Listing of the National Vulnerability Database*. Web document. URL: https://nvd.nist.gov/vuln/full-listing (visited on 2022-09-23).

[2] Úlfar Erlingsson, Yves Younan, and Frank Piessens. "Low-Level Software Security by Example." In: *Handbook of Information and Communication Security*. Ed. by Peter Stavroulakis and Mark Stamp. Berlin/Heidelberg, Germany: Springer, 2010, pp. 633–658. ISBN: 978-3-642-04117-4. DOI: 10.1007/978-3-642-04117-4_30.

[3] László Szekeres et al. "SoK: Eternal War in Memory." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[4] Qualcomm Technologies, Inc. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. Tech. rep. San Diego, CA, USA, 2017.

[5] Gabriele Serra et al. "PAC-PL: Enabling Control-Flow Integrity with Pointer Authentication in FPGA SoC Platforms." In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 241–253. DOI: 10.1109/RTAS54340.2022.00027.

[6] Ulrich Drepper. "What Every Programmer Should Know About Memory." 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[7] Paul Mckenney. "Memory Barriers: a Hardware View for Software Hackers." 2010. URL: http://www.puppetmastertrading.com/images/hwViewForSwHackers.pdf.

[8] C++ reference. *std::memory_order*. Web document. 2022. URL: https://en.cppreference.com/w/cpp/atomic/memory_order (visited on 2022-09-23).

[9] Arm Limited. *Arm® Architecture Reference Manual for A-Profile Architecture*. DDI 0487H.a. Cambridge, England, 2022.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Order Number: 325462-076US. Santa Clara, CA, USA, 2021.

[11] IEEE and The Open Group. *IEEE Std 1003.1TM-2017 (POSIX.1-2017)*. Web document. 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 2022-09-23).

[12] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection." In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380.

[13] Aleph One. "Smashing the Stack for Fun and Profit." In: *Phrack* 7.49 (1996). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[14] Microsoft. *Data Execution Prevention*. Web document. 2022-08-02. URL: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention (visited on 2022-09-23).

[15] Jonathan Corbet. *x86 NX support*. LWN.net article. 2004-06-02. URL: https://lwn.net/Articles/87814/ (visited on 2022-09-23).

[16] Alexander Peslyak. *Getting around non-executable stack (and fix)*. Web document. 1997-08-10. URL: https://seclists.org/bugtraq/1997/Aug/63 (visited on 2022-09-23).

[17] 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*. CCS '07. Alexandria, VA, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313.

[18] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications." In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012). ISSN: 1094-9224. DOI: 10.1145/2133375.2133377.

[19] PaX. *Address Space Layout Randomization*. Web document. 2001. URL: https://pax.grsecurity.net/docs/aslr.txt (visited on 2022-09-23).

[20] Raoul Strackx et al. "Breaking the Memory Secrecy Assumption." In: *Proceedings of the Second European Workshop on System Security*. EUROSEC '09. Nuremburg, Germany: Association for Computing Machinery, 2009, pp. 1–8. ISBN: 9781605584720. DOI: 10.1145/1519144.1519145.

[21] The GNU project. *Using the GNU Compiler Collection (GCC)*. Version 10.2. 2022. URL: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/.

[22] Perry Wagle, Crispin Cowan, and Immunix. "StackGuard: Simple Stack Smash Protection for GCC." In: *GCC Developers Summit*. 2004.

[23] Jake Edge. *"Strong" stack protection for GCC*. LWN.net article. 2014-02-05. URL: https://lwn.net/Articles/584225/ (visited on 2022-09-23).

[24] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker." In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA, USA: USENIX Association, 2012, p. 28.

[25] Martín Abadi et al. "Control-Flow Integrity." In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165.

[26] Ali Jose Mashtizadeh et al. "CCFI: Cryptographically Enforced Control Flow Integrity." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, CO, USA: Association for Computing Machinery, 2015, pp. 941–951. ISBN: 9781450338325. DOI: 10.1145/2810103.2813676.

[27] Roberto Maria Avanzi. "The QARMA Block Cipher Family." 2016. URL: https://eprint.iacr.org/2016/444.pdf.

[28] Hans Liljestrand et al. "PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication." In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 177–194. ISBN: 9781939133069.

[29] Joseph Ravichandran et al. "PACMAN: Attacking ARM Pointer Authentication with Speculative Execution." In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022. ISBN: 9781450386104. DOI: 10.1145/3470496.3527429.

[30] The GNU project. *GNU Compiler Collection (GCC) Internals*. Visited on 2022-09-23. 2022. URL: https://gcc.gnu.org/onlinedocs/gccint/.

[31] The GNU Project. *GCC 4.5 Release Series: Changes, New Features, and Fixes*. Web document. 2010-04-14. URL: https://gcc.gnu.org/gcc-4.5/changes.html (visited on 2022-09-23).

[32] Volodymyr Kuznetsov et al. "Code-Pointer Integrity." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO, USA: USENIX Association, 2014, pp. 147–163. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[33] Isaac Evans et al. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity." In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 781–796. DOI: 10.1109/SP.2015.53.

[34] Yu Wang et al. "RetTag: Hardware-Assisted Return Address Integrity on RISC-V." In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec '22. Rennes, France: Association for Computing Machinery, 2022, pp. 50–56. ISBN: 9781450392556. DOI: 10.1145/3517208.3523758.

[35] Ruan de Clercq and Ingrid Verbauwhede. *A survey of Hardware-based Control Flow Integrity (CFI)*. 2017. DOI: 10.48550/ARXIV.1706.07257.

[36] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity." In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337175.

[37] George Christou et al. "Hard Edges: Hardware-Based Control-Flow Integrity for Embedded Devices." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, Proceedings*. Samos, Greece: Springer-Verlag, 2021, pp. 275–287. ISBN: 978-3-031-04579-0. DOI: 10.1007/978-3-031-04580-6_18.

[38] Hans Liljestrand et al. "Protecting the Stack with PACed Canaries." In: *Proceedings of the 4th Workshop on System Software for Trusted Execution*. SysTEX '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019. ISBN: 9781450368889. DOI: 10.1145/3342559.3365336.

[39] Hans Liljestrand et al. "PACStack: an Authenticated Call Stack." In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021-08, pp. 357–374. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[40] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. "PTAuth: Temporal Memory Safety via Robust Points-to Authentication." In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021-08, pp. 1037–1054. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[41] Rémi Denis-Courmont et al. "Camouflage: Hardware-Assisted CFI for the ARM Linux Kernel." In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC '20. Virtual Event, USA: IEEE Press, 2020. ISBN: 9781450367257.

[42] Arm Limited. *Procedure Call Standard for the Arm® Architecture*. 2022Q1. Cambridge, England, 2022. URL: https://github.com/ARM-software/abi-aa.

[43] H. J. Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. Version 1.0. 2022-08-27. URL: https://gitlab.com/x86-psABIs/x86-64-ABI.

[44] The Linux kernel contributors. *Kconfig Language*. Linux kernel documentation. 2022. URL: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.rst (visited on 2022-09-23).

[45] Yann Collet. *xxHash fast digest algorithm*. Web document. 2018-10-10. URL: https://github.com/Cyan4973/xxHash/blob/dev/doc/xxhash_spec.md (visited on 2022-09-23).

[46] Jonathan Corbet. *DebugFS*. Linux kernel documentation. 2009. URL: https://docs.kernel.org/filesystems/debugfs.html (visited on 2022-09-23).

[47] The Linux kernel contributors. *The kernel's command-line parameters*. Linux kernel documentation. 2022. URL: https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.rst (visited on 2022-09-23).

[48] Ulrich Drepper. "ELF Handling For Thread-Local Storage." 2013. URL: https://www.akkadia.org/drepper/tls.pdf.

[49] Florian Rommel et al. "From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes." In: *14th Symposium on Operating System Design and Implementation (OSDI '20)*. 2020, pp. 651–666. URL: https://www.usenix.org/conference/osdi20/presentation/rommel.

[50] Daniel Gruss et al. "KASLR is Dead: Long Live KASLR." In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Cham: Springer International Publishing, 2017, pp. 161–176. ISBN: 978-3-319-62105-0.

[51] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[52] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN: 0596005652.

[53] The Linux man-pages project. *cpuset(7)*. Linux manual page. 2022. URL: https://man7.org/linux/man-pages/man7/cpuset.7.html (visited on 2022-09-23).

[54] The Mitre Corporation. *CVE® Program Mission*. Web document. URL: https://www.cve.org (visited on 2022-09-23).

[55] Heiko Falk et al. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 2:1–2:10. ISBN: 978-3-95977-025-5. DOI: 10.4230/OASIcs.WCET.2016.2. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6895.

[56] The Linux kernel contributors. *NO_HZ: Reducing Scheduling-Clock Ticks*. Linux kernel documentation. 2022. URL: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt (visited on 2022-09-23).

[57] Intel Corporation. *Breakthrough AES Performance with Intel® AES New Instructions*. Tech. rep. Santa Clara, CA, USA, 2010.

[58] Dormando. *About Memcached*. Web document. URL: https://memcached.org/about (visited on 2022-09-23).

[59] Redis Labs. *Memtier: NoSQL Redis and Memcache traffic generation and benchmarking tool*. Web document. URL: https://github.com/RedisLabs/memtier_benchmark (visited on 2022-09-23).

[60] The Linux kernel contributors. *Why the "volatile" type class should not be used*. Linux kernel documentation. 2022. URL: https://www.kernel.org/doc/Documentation/process/volatile-considered-harmful.rst (visited on 2022-09-23).