國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master’s Thesis

利用 ARM 指標認證與棧回溯技術以保護系統呼叫及控制流

Utilize Arm Pointer Authentication and Stack Unwinding to Protect System call Usage and Control Flow

許智凱

Chih-Kai Hsu

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li, Ph.D.

中華民國 113 年 8 月

August 2024

doi:10.6342/NTU202402925

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

利用 ARM 指標認證與棧回溯技術以保護系統呼叫及控制流

Utilize Arm Pointer Authentication and Stack Unwinding to Protect System call Usage and Control Flow

本論文係許智凱（學號 R11944008）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國 113 年 7 月 11 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Graduate Institute of Networking and Multimedia on 11 July 2024 have examined a Master's Thesis entitled above presented by HSU, CHIH-KAI (student ID: R11944008) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

黎士瑋	陳君明	廖世偉
（指導教授 Advisor）		

系（所）主管 Director: 鄭卜壬

doi:10.6342/NTU202402925

致謝

首先，我要感謝我的指導教授黎士瑋博士，您在整個碩士班研究過程中給予了我極大的幫助和指導，讓我受益匪淺。您的建議不僅推動了我的研究進展，還提供了比我自己更全面和有條理的分析角度，讓我在評斷問題時能有更全面的思考。在這幾個月的論文撰寫過程中，您不厭其煩地審閱我的論文，指出文字中的邏輯謬誤和解釋不清之處，使我的學術寫作和邏輯闡述能力得到了顯著提升。

本論文的完成還要感謝廖世偉教授和陳君朋教授擔任我的口試委員。你們的建議和意見使我的論文更加完整和嚴謹。

我也要感謝我的家人和朋友們，正是因為有你們的支持，我才能完成這篇論文。

此外，特別感謝 OpenAI 的 ChatGPT 幫助我改進了論文的英文表達，許多論文中的字句（包括本致謝）得益於它的幫助，才能如此通順、清晰和流暢。

謝謝。

許智凱

國立臺灣大學資訊網路與多媒體研究所

中華民國一百一十三年七月

doi:10.6342/NTU202402925

摘要

現代系統為應用程式提供各種服務，這些服務主要通過系統調用訪問。系統調用經常被利用於嚴重攻擊中，例如控制流劫持攻擊。因此，與安全相關的系統調用（如 mprotect、mmap 和 execve）在整個攻擊鏈中起著關鍵作用。另一方面，ARM 處理器現在越來越多地部署在桌面和數據中心。雖然先前的研究已經構建了保護 x64 架構上系統調用使用的防禦機制，但我們提出了一種新穎的框架，以確保內存不安全編程語言（C/C++）在 ARM 架構上的系統調用使用的安全性。

我們確保合法的系統調用使用具有以下屬性：**系統調用調用的控制流完整性**。首先，我們在 Linux 內核中引入了一個基於堆棧回溯的監控器。其次，我們利用 ARMv8.3 處理器中可用的指針驗證（PA）功能來保護控制流敏感的指針，如函數指針和 C++ 虛表指針。通過這些防禦機制，我們可以有效地破壞攻擊鏈，防止攻擊者達成他/她的目標。

我們的框架由兩個主要組件組成：1）可加載內核模塊（LKM）和 2）定制的 LLVM 編譯器。我們的安全案例研究表明，我們可以有效地擊敗所有攻擊，包括真實世界的漏洞利用。我們使用三個常見的系統調用密集型程序（Lighttpd、NGINX 和 SQLite）以及 SPEC CPU2017 基準套件來評估性能。結果顯示，Lighttpd 的性能開銷為 0.68%，NGINX 為 0.45%，而 SPEC CPU2017 基準套件的平均開銷為 2.95%。我們在 Section 6.2.3 中解釋了 SQLite 開銷較高的原因。

doi:10.6342/NTU202402925

關鍵字：系統呼叫、棧回溯、指標認證、控制流完整性

doi:10.6342/NTU202402925

Abstract

Modern systems provide various services to applications, primarily accessed through system calls. System calls are frequently utilized in serious attacks, such as control-flow hijacking attack. Therefore, security-related system calls, such as `mprotect`, `mmap` and `execve` play a pivotal role in the entire attack chain. On the other hand, ARM processors are increasingly deployed on desktops and in data centers nowadays. While previous works have built defense mechanisms to protect system call usages on x64 architecture, we propose a novel secure properties for system call usages for memory-unsafe programming languages (C/C++) on ARM architecture.

We ensure a property for legitimate system call usage: **the control flow integrity of system call invocations**. Firstly, we introduce a stack unwinding-based `monitor` in the Linux kernel. Secondly, we utilize the Pointer Authentication (PA) feature available in ARMv8.3 processors to protect control-flow-sensitive pointers, such as function pointers and C++ Vtable pointers. With these defense mechanisms, we can effectively corrupts the

doi:10.6342/NTU202402925

attack chain, preventing the attacker from achieving her goals.

Our framework consists of two main components 1) a loadable kernel module (LKM) and 2) a customized LLVM compiler. Our security case study demonstrates that we can effectively defeat all attacks, including real-world exploits. We evaluate the performance using three popular system call-intensive programs: Lighttpd, NGINX, and SQLite, as well as the SPEC CPU2017 benchmark suite. Our results indicate an overhead of 0.68% for Lighttpd, 0.45% for NGINX, and an average of 2.95% for the SPEC CPU2017 benchmark suite. We explain the reasons for the higher overhead on SQLite in the Section 6.2.3.

Keywords: System Call, Stack Unwind, Pointer Authentication, Control Flow Integrity

doi:10.6342/NTU202402925

Contents

doi:10.6342/NTU202402925

doi:10.6342/NTU202402925

doi:10.6342/NTU202402925

List of Figures

doi:10.6342/NTU202402925

List of Tables

doi:10.6342/NTU202402925

Chapter 1 Introduction

Modern applications are designed with a myriad of functionalities to cater to diverse user needs. These functionalities rely on system calls, a set of services provided by the OS kernel. While system calls facilitate seamless communication between applications and the OS kernel, improper usage can lead to significant security issues. Attackers often exploit program bugs to invoke system calls for malicious purposes, such as downloading and executing harmful payloads.

Various approaches have been proposed to address system call security issues. Some methods focus on reducing the program's attack surface. For example, debloating techniques [7, 42, 43] aim to eliminate unused code that might invoke system calls. Other approaches [16, 17, 20] involve profiling program behaviors to filter out unused system calls. However, these strategies cannot prevent the execution of sensitive system calls (e.g., execve) essential to the program's functionality.

Arm processors have been increasingly adopted across a wide range of platforms, including mobile and embedded devices, personal computers, automobiles, and cloud servers [4, 49]. This widespread adoption underscores the critical need to secure system call usage in programs running on Arm-based platforms. In this thesis, we propose a novel framework designed to protect legitimate system call usage in C/C++ programs on

doi:10.6342/NTU202402925

Arm-based platforms. Our framework ensures a key property: **the control flow integrity of system call invocations**. Securing control flow integrity (CFI) for all control transfers can introduce significant performance overhead. Therefore, we focus specifically on ensuring CFI for direct and indirect function calls. For direct function calls, we incorporate a stack unwinding-based approach. We implement a secure `monitor` that hooks into Linux's sensitive system calls. When a task invokes a sensitive system call, the `monitor` unwinds the caller's call stack to verify whether the call was made through a legitimate control-flow function call path. To simplify deployment efforts, we leverage the program's DWARF [1] debug information, which is readily available in the program's .eh_frame section. This approach enables us to perform call stack unwinding without requiring program recompilation, thus facilitating easier integration and deployment.

We utilize the Pointer Authentication (PA) [32, 44] feature available from ARMv8.3 processors to secure the control flow integrity of indirect function calls. PA allows us to tag a function pointer and store the tag in the unused bits of the pointer. At function pointer usage sites, we validate the tag. If a compromise is detected, the hardware aborts the program to prevent the use of the corrupted pointer. To protect the integrity of indirect calls, we instrument programs to tag and validate function pointers. Additionally, we extend this protection to C++ programs by tagging pointers to the virtual method table (Vtable). This ensures the integrity of virtual method invocations, safeguarding against Vtable hijacking and related attacks.

We leverage both runtime and static approaches to collectively enforce the intended system call security properties. Our framework consists of a loadable kernel module and LLVM passes. The kernel module implements our `monitor`, while the LLVM passes facilitate PA-based pointer protection. Although we aim to provide a comprehensive so-

doi:10.6342/NTU202402925

lution for securing system call usage, the individual components (e.g., the `monitor` and compiler) offer standalone system call protection features that are independent yet complementary. Developers can configure and activate features according to their specific needs. For example, Arm platforms requiring higher performance could employ only our `monitor` to harden the system without necessitating program instrumentation. This flexibility allows for tailored security measures based on the performance and security requirements of different deployment scenarios.

We have constructed a prototype aimed at safeguarding system call usage in Arm-based programs. This prototype is compatible with commonly used network-facing server applications running on standard Linux distributions deployed across embedded, desktop, and server Arm hardware. Through our evaluation, we have demonstrated the effectiveness of our framework in defending against vulnerabilities that exploit control-flow hijacking techniques to facilitate malicious system call invocations. Additionally, we conducted performance evaluations on protected applications. Our findings indicate that our framework imposes only modest overhead, ranging from 0.45% to 4.22% without SQLite, when compared to unprotected applications. This minimal performance impact underscores the efficiency and practical viability of our approach in enhancing system call security without significantly compromising system performance. On the other hand, we explain the reasons for the higher overhead on SQLite in the Section 6.2.3.

doi:10.6342/NTU202402925

Chapter 2 Background

2.1 Stack Layout Information for binaries

2.1.1 Frame Pointer and Stack Pointer

The stack pointer indicates the top of the current function's stack, while the frame pointer points to the top of the caller's stack, which contains the return address for the current function. On ARM64 architecture, the stack pointer is held in the `sp` register, and the frame pointer is held in the `x29` register.

2.1.2 .eh_frame

.eh_frame is a section in an ELF binary. It contains the program's call stack trace information generated by the GNU C tool-chain in the DWARF[1] format. The .eh_frame consists of the stack layout information, such as the PC relative address offset to the stack top and frame record. The information is helpful for stack unwinding (to retrieve the function's return address) during signal handling, exception handling, and debugging. The .eh_frame section comprises one or more FDEs (Frame Description Entry), and each FDE contains a list of CFIs (Call Frame Instructions). Each FDE is associated with a particular function or section of code within the object file. In FDE, two entries, pc begin and

doi:10.6342/NTU202402925

pc range, define the begin address and the range of the current function. CFI gives us information about restoring the return address register and callee-saved register.

We show the Arm64 assembly code in ARM64 and the FDE of a function `ngx_alloc` in Nginx in Listing 1 and Listing 2, respectively. In Listing 2, line 2 shows the range of the function, which is from `0x408a4` to `0x40908`, and each row from line 4 to line 8 is decoded from one or more CFIs. CFA (Canonical Frame Address) stores the value of the stack pointer at the call site in the previous frame. Based on the CFA value, one can derive the addresses where specific registers and the return address are stored in the stack. For instance, line 6 from Listing 2 specifies that while the PC is set to the address `0x408b4`, the CFA points to the address `sp + 48` and the callee's return address (from the `ra` column) is stored at the address `c - 40` (i.e., `sp + 8`). This information facilitates stack unwinding.

```
00000000000408a4 <ngx_alloc>:

   408a4:   a9bd7bfd    stp     x29, x30, [sp, -48]!
   408a8:   910003fd    mov     x29, sp
   408ac:   a90153f3    stp     x19, x20, [sp, 16]
   408b0:   f90013f5    str     x21, [sp, 32]
   408b4:   aa0003f5    mov     x21, x0
   408b8:   aa0103f4    mov     x20, x1
```

Listing 1: assembly code of `ngx_alloc`

```
1 00005780 0000000000000028 00005784 FDE cie=00000000
2 pc=408a4..40908
3 LOC     CFA     x19   x20   x21   x29   ra
4 408a4   sp+0    u     u     u     u     u
5 408a8   sp+48   u     u     u     c-48  c-40
6 408b4   sp+48   c-32  c-24  c-16  c-48  c-40
7 408d8   sp+0    u     u     u     u     u
8 408dc   sp+48   c-32  c-24  c-16  c-48  c-40
```

Listing 2: FDE of `ngx_alloc`

doi:10.6342/NTU202402925

2.2 Stack Unwinding

Stack unwinding is the process of deallocating or ”unwinding” function call frames from the call stack during program execution. When a function is called, its execution context, including local variables and parameters, is pushed onto the call stack. As functions return, their respective frames are removed from the stack in a last-in, first-out (LIFO) fashion.

Stack unwinding typically occurs when a function returns or when an exception is thrown. During unwinding, the runtime system executes destructors for local objects within each frame as they are removed from the stack. This ensures that resources held by local variables, such as memory or file handles, are properly released.

In languages with built-in exception handling mechanisms, stack unwinding is an essential part of exception propagation. When an exception is thrown, the runtime system unwinds the stack until an appropriate exception handler is found or until the program terminates if no handler is available. During unwinding, destructors are called for objects in each frame to perform cleanup operations.

2.3 Arm Pointer Authentication

Arm introduced Pointer Authentication (PA) [32, 44] for Armv8.3-A processors. PA aims to protect the integrity of pointers with minimal impact on performance and memory usage. PA provides instructions to sign a pointer, i.e., to generate a Message Authentication Code (MAC) for a pointer called the Pointer Authentication Code (PAC), and authenticate the PAC. Arm PA utilizes the QARMA [12] cipher. PA provides five keys,

doi:10.6342/NTU202402925

two for each data code pointer and one generic user key. The keys are stored in privileged hardware registers that users cannot access. PA provides instruction prefixed with `pac` and `aut`, respectively, for signing and authenticating, followed by two characters that specify the key to use. For instance, the `pacia` instruction signs a code pointer with the A-key. To create PACs, users send two 64-bit values, a pointer and the modifier, to the signing instructions. The QARMA cipher uses the instruction's associated key to produce and place the resulting PAC into the upper unused bits of the 64-bit pointer (Figure 2.1(a)). The placement renders the signed pointer unusable — accessing the pointer causes a fault in address translation. Users use the `aut` instruction respective to the signing key to authenticate the pointer. The same modifier used when signing the pointer is also used when authenticating it (Figure 2.1(b)). PA validates the pointer, i.e., the recomputed PAC matches the one stored in the pointer. If the integrity is validated, PA removes the PAC from the pointer; otherwise, PA leaves the pointer unusable.

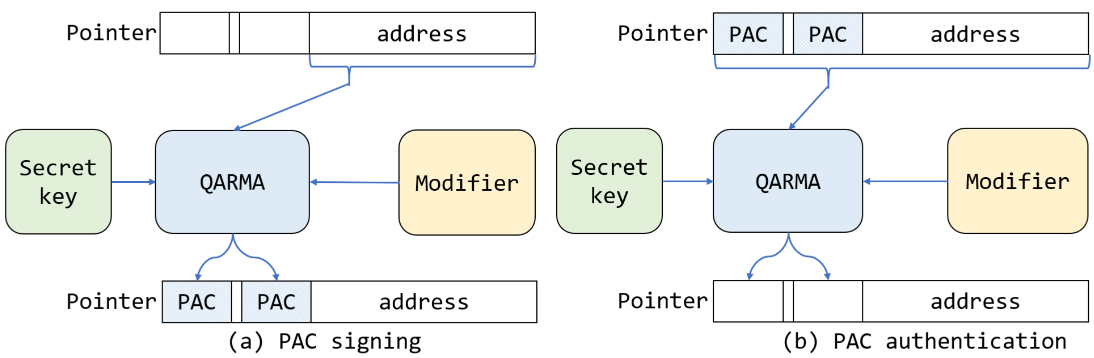

Figure 2.1: PAC signing and authentication

2.4 Code Reuse Attack

Code reuse attacks are a type of exploit where an attacker uses existing code within a program to execute arbitrary actions, circumventing traditional security measures like

doi:10.6342/NTU202402925

non-executable memory protections, such as XN [11] on Arm architecture. Instead of injecting new, potentially detectable malicious code, attackers repurpose fragments of the program's own code, making these attacks harder to detect and prevent.

Return-Oriented Programming (ROP) is a sophisticated and common form of code reuse attack. In an ROP attack, an attacker manipulates the control flow of a program to execute a sequence of short instruction sequences, known as "gadgets," which already exist in the program's memory. Each gadget ends with a return instruction (RET). By chaining these gadgets together, attackers can perform complex operations without introducing any new code.

2.5 Virtual Functions in C++

In C++, dynamic polymorphism is achieved through virtual functions managed by a virtual table (VTable). The compiler builds a VTable for each class that contains virtual functions. The VTable contains entries for each of the class' virtual functions that stores a pointer to the function's address in memory. A VTable entry could point to a virtual function defined by the class or inherited from a base class. The compiler creates a virtual table pointer (VPointer) for an instantiated class object that points to the object's respective VTable.

When a virtual function is called through a pointer or reference to a base class object, the compiler generates code to perform a virtual function dispatch. Consider the example in Listing 3, this involves looking up the correct function pointer in the object's VTable based on the actual type of the object at runtime. Once the correct function pointer is found, the corresponding function is invoked.

doi:10.6342/NTU202402925

```
ldur    x0, [x29, #-16]    ; load the object
ldr     x8, [x0]           ; load VPointer
ldr     x8, [x8, #8]       ; find the correct function using offset
blr     x8
```

Listing 3: Assembly Code for C++ Virtual Function Dispatch

2.6 VTable Hijacking Attacks

In C++ programs, VTables are located in a read-only memory region. Instead of compromising VTable entries, attackers focus on exploiting vulnerabilities to corrupt VPointers to point to a crafted VTable that contains malicious pointers to hijack the program's control flow.

Type confusion attacks occur when an object is cast to an invalid type, often with a different size than the underlying object. This allows attackers to access unintended memory and potentially overwrite VPointers. Such attacks are typically executed through an illegal downcast forced by attackers, such as casting to a sibling class.

Counterfeit Object-Oriented Programming (COOP) [46] creates objects with counterfeit (fake or synthetic) types. It involves manipulating the internal structure of objects, particularly VTables, to create objects that appear to be of one type but are actually of another type. In COOP attacks, attackers exploit vulnerabilities related to type confusion or memory corruption to manipulate object layouts and VPointers. By doing so, they can trick the program into treating these counterfeit objects as legitimate instances of a different type.

doi:10.6342/NTU202402925

Chapter 3 Threat Model and Assumptions

We assume a powerful adversary who can read and write arbitrary memory by exploiting memory vulnerabilities such as heap or stack overflows in a program. The targeted system employs Execute Never [11] (XN) and Address Space Layout Randomization [40] (ASLR) to prevent attackers from injecting or modifying code. The hardware and OS kernel are trusted, and side-channel attacks are not considered in this scope.

This thesis focuses on protecting system calls. Our analysis indicates that attackers often aim to exploit system calls to interact with the host operating system and carry out their attacks. For instance, Goktas et al. [21] demonstrated that a code reuse attack could change the permissions of an existing memory area (e.g., using `mprotect` on Linux), allowing code injection and bypassing XN protection. Therefore, we target and secure system calls to significantly hinder attackers' ability to achieve their malicious goals. Specifically, we concentrate on system calls related to security actions, as listed in Table 3.1.

doi:10.6342/NTU202402925

Table 3.1: Protected system call set.

System call category	Available System calls
Arbitrary Code Execution	`execve`, `execveat`, `clone`
Memory Permission	`mprotect`, `mmap`, `mremap`
Privilege Escalation	`chmod`, `setuid`, `setgid`, `setreuid`
Networking	`socket`, `bind`, `connect`, `listen`, `accept`, `accept4`
File system-related	`openat`, `read`, `write`, `readv`, `writev`, `sendfile`, `recvfrom`

doi:10.6342/NTU202402925

Chapter 4 Design

We aim to protect an application from attacks that exploit sensitive system calls through code reuse attacks. Therefore, we aim to protect the usage of sensitive system calls. To achieve the goal, we focus on protecting the property: the control flow integrity of system call invocations.

4.1 Control Flow Integrity of System Call Usage

We protect the control flow integrity of direct and indirect function calls. For the former, we leverage a stack unwinding-based approach to enforce the integrity of a function call path that leads to system call invocation. We use Arm's PA to protect attackers from corrupting call targets of indirect function calls and C++ VPointers.

4.1.1 Backward-Edge CFI Protection

We make **two observations**. First, for a legitimate system call invocation via direct function calls, a function call path must exist from the `main` function to the function that makes the system call. Second, for a direct function call, the memory address before the saved return address must contain a call instruction (e.g., `bl` instruction in Arm64).

doi:10.6342/NTU202402925

We rely on a secure `monitor` module to protect backward-edge CFI of system call usage. The `monitor` hooks Linux's system call table to interpose the invocation of sensitive system calls. It performs algorithm 1 to validate the legality of the function call path that leads to the system call invocation. At a level, the algorithm checks if the observations are satisfied in the function call path that leads to the system call usage. If not, the `monitor` uncovers that an attacker has corrupted the stack to hijack the program's control flow to invoke a sensitive system call.

The `monitor` acquires the stack layout information of a program from the program's `.eh_frame` section. When the system call is invoked, the `monitor` hooks `execve` to retrieve a program's `.eh_frame`. The `monitor` also acquires the program's shared object dependencies to support unwinding shared libraries. When a sensitive system call is later invoked, the `monitor` first uses the call site's PC to get the corresponding CFA. The `monitor` then uses the CFA to get the saved return address of the call frame. For each function in the call path, the `monitor` checks two conditions: (1) if the instruction located before the saved return address in the function's caller is a call instruction, and (2) if the target of the call instruction is the start of the function. If either condition is violated, the unwinder concludes that an attack has corrupted the stack and skips the system call invocation. We denote the instruction in the rest of the paper as *call site checking*. Call site checking is conducted every time the `monitor` derives the return address for a call frame.

As shown in algorithm 1, the unwinding process includes a `while` loop (line 2) that iterates each call frame. In the loop body, it first retrieves the `.eh_frame` by the current program counter (PC). If this retrieval is unsuccessful, the `monitor` attempts to find the virtual memory area (VMA) by PC and then get the `.eh_frame` using VM_FILE in the VMA's structure (line 4 to 12). Next, the `monitor` gets the current function's entry point

doi:10.6342/NTU202402925

Algorithm 1: Unwinder

```
Function Unwinding:
    while true do
        PC = REGS → PC; EhFrame = GetEhFrame(PC)
        if EhFrame does not exist then
            VMA = GetVMA(PC)
            if VMA is invalid or does not exist then
                Abort the process
            end
            if VMA → VM_FILE does not exist then
                Abort the process
            end
            EhFrame = GetEhFrameFromVMA(VMA)
        end
        EntryPoint = GetEntryPoint(EhFrame, PC)
        if EntryPoint is terminate function then
            return Success
        end
        CFI = ProcessCFI(EhFrame)
        if CFI is invalid then
            Abort the process
        end
        REGS = UpdateREGS(REGS, CFI)
        RA = REGS → X30
        CallInsn = DisassembleInsn(RA - 4)
        if CallInsn is really a call instruction then
            Callee = GetCalleeFromCallInsn(CallInsn)
            if Callee is not the same as EntryPoint then
                Abort the process
            end
        else
            Abort the process
        end
        REGS → PC = RA
    end
```


doi:10.6342/NTU202402925

by the `.eh_frame` and PC (line 14). If the current function's entry point is the terminate function (e.g., `main`), it concludes that the call path is valid; otherwise, the `monitor` processes the CFI from the `.eh_frame` and restore register according to the CFI (line 18 to 22). By the way, we cached processed CFI to accelerate the unwinding process. After that, the `monitor` first gets the saved return address from the restored registers (line 23), disassembles the instruction stored in the address *RA - 4* (line 24), and performs call site checking (line 25 to 32). During the unwinding process, the `monitor` does not update the actual hardware registers (i.e., update stack and frame pointer and callee-saved registers) and execute destructors for local objects within each frame from the saved context in the stack. Doing this will interfere with the program's execution. Instead, the `monitor` mocks register updates to a pseudo register context allocated from memory.

4.1.2 Forward-Edge CFI Protection

As the above section mentioned, we checked the integrity of entire call chain through unwinding and call site checking, however, if attackers hijack a function pointer to tamper the control flow of a program, the layout of stack would not be corrupted, consequently, this type of attack can bypass our unwinding mechanism. More specific, unwinding can still back to the entry point of the program. On the other hand, because `.eh_frame` can not restore all registers and registers' values are not guaranteed any tampering. Therefore, our call site checking mechanism only works for `bl` instruction, namely, direct function call.

We protect forward-edge control flow integrity to prevent attackers from hijacking indirect function calls. We utilize Arm's Pointer Authentication (see Section 2.3) to effectively protect the integrity of function pointers. We introduced a new LLVM pass to

doi:10.6342/NTU202402925

instrument PA instructions to the program. The LLVM pass instruments PA's signing instructions before storing an address value to (1) a function pointer or (2) the VPointer of a class object. It also instruments PA's authentication instructions before the signed pointers are used for control transfers. We focus on protecting VPointers because VTables are already set to read-only.

Selecting PA modifier Previous work that built on PAC to protect pointer integrity either uses a constant modifier [10] for all signed pointers or assigns a unique 64-bit modifier [23] for each pointer to be protected by PA. Because PA is vulnerable to pointer reuse attacks [23, 28] where an authenticated pointer is replaced with another with the same modifier. Therefore, using a constant modifier increases the scope of this substitution. On the other hand, the latter approach results in overhead in storing modifiers and managing the associated metadata. Unlike these approaches, we proposed using a function's signature as its associated PA modifier to enhance efficiency. The signatures of system calls and a user program's custom function pointers frequently differ. Moreover, we observed that an attacker generally aims to replace an existing function pointer with the address of an intended target function with a different signature.

We utilize the address of the VPointer as a modifier for signing the VPointer, binding the pointer's location and value. Because VPointer do not change much over the program's lifetime, binding pointers to addresses ensures that the pointer value remains unmodified, effectively preventing that location from being corrupted.

doi:10.6342/NTU202402925

Chapter 5 Implementation

We implemented a prototype based on the proposed design. The prototype comprises a loadable `monitor` kernel module and an extended LLVM compiler. The resulting kernel module consists of 4,956 lines of code (LoC). We built the kernel module independently from the Linux kernel source to enhance deployability. The module supports Asahi Linux 6.3.0-11 and mainline Linux 6.3-rc7. We built on LLVM 16.0.0, modified its front-end, i.e., Clang, and extended LLVM's AArch64 backend. In addition, we added a LLVM pass for forward-control-flow protection based on Arm PA. The changes for LLVM in total require 1,668 LoC.

Figure 5.1 illustrates the overall architecture of LLVM. The modified Clang front-end identifies accesses to C++ VPointers and prepares the PA modifiers. The optimization pass identifies function pointers, generate the necessary metadata for PA modifiers, and prepare initializers for statically allocated pointers. Finally, the backend pass retrieves the PA modifiers and instruments the appropriate low-level instructions.

We have introduced our definition of sensitive system calls in chapter 3. Additionally, we extend our protections to file system-related system calls to guard against information disclosure attacks. By securing both the sensitive system calls and those related to the file system, we aim to mitigate a broad range of potential attack vectors.

doi:10.6342/NTU202402925

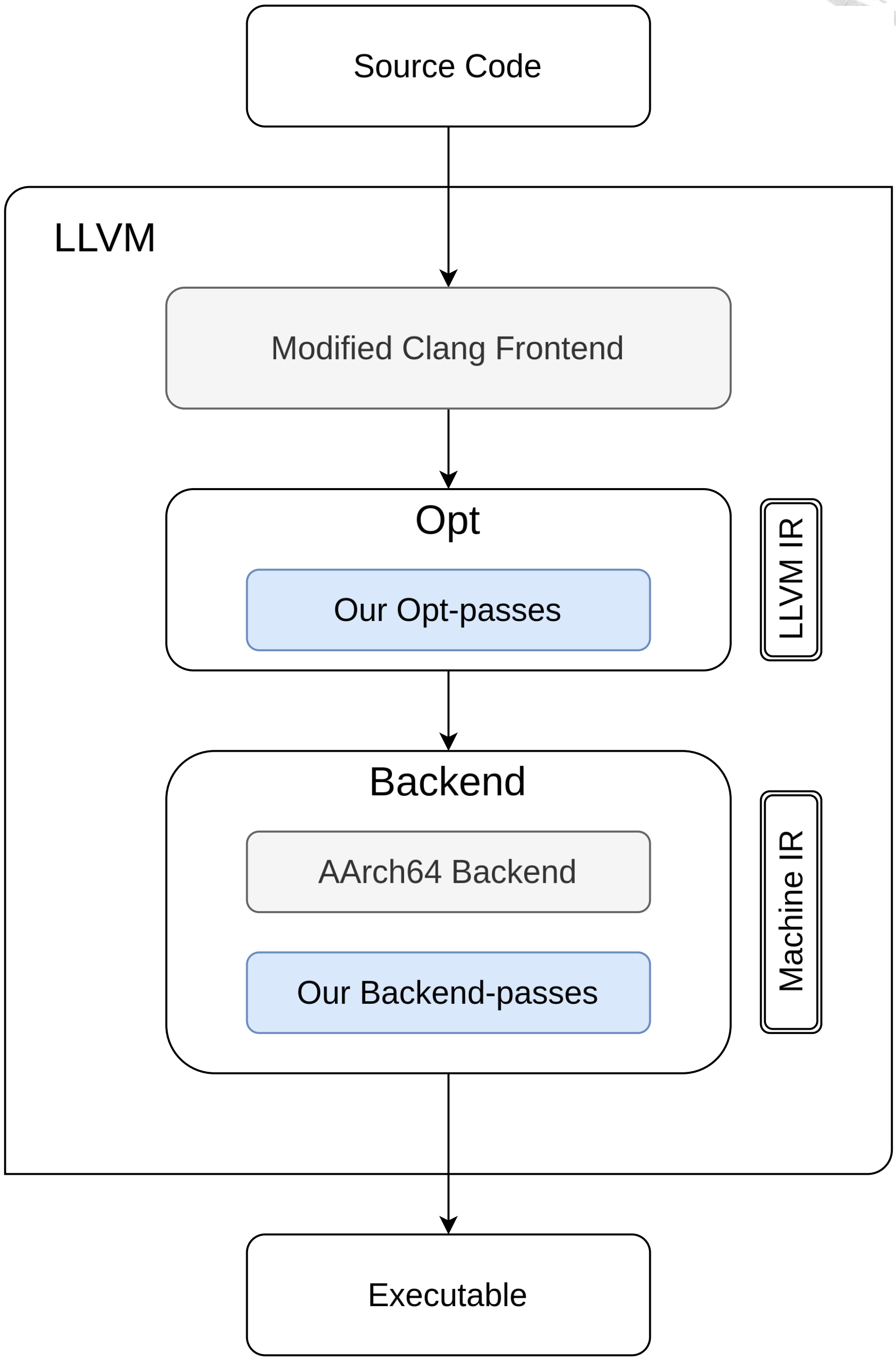

Figure 5.1: LLVM architecture

doi:10.6342/NTU202402925

5.1 Loadable Kernel Module

In this section, we describe how stack unwinding and call site checking are implemented within a loadable kernel module. Initially, the kernel module retrieves essential metadata from the launching executable, including the contents of the `.eh_frame` section, the terminate function (commonly referred to as `main`), and any dependencies (i.e., used shared objects). When a protected system call is invoked, the kernel module retrieves this metadata and executes Algorithm 1 to validate the legality of the system call's control flow. If either the unwinding process or the call site checking fails, the process is promptly aborted to prevent further execution and potential security breaches.

5.1.1 Intercept system call

When invoking system calls, we need to intercept the system calls and perform stack unwinding and call site checking. To this end, we hook into the system call table in the Linux kernel. However, since we are implementing our mechanism as a loadable kernel module, we may encounter limitations in accessing certain kernel symbols, such as the system call table. To overcome this limitation, we utilize Kernel Probes (KProbes) to locate the missing symbols. Once we have identified the necessary symbols, we can proceed to replace the entries in the system call table accordingly. Within the hooked system call function, we integrate our validation mechanism. If the validation is successful, we proceed to invoke the original system call. However, if the validation fails, we abort the process immediately.

doi:10.6342/NTU202402925

5.1.2 Mechanisms

The `monitor` module reads the section table [30] from the program's binary and then locates the load memory address of the `.eh_frame` section and its section size. The module also locates the program's string table (`.strtab`) and symbol table (from the `.symtab` section) from the section table. Both are used to resolve the address of the termination function for unwinding, i.e., the `main` function. The `monitor` supports unwinding and call site checking against dynamically linked program binaries. When an `execve` system call is invoked, the `monitor` examines whether the program to be executed contains a `.dynamic` section [29]. If such a section is present, the module traverses the `.dynamic` section to identify all shared libraries that the program is linked with. The `monitor` searches for these shared library objects from pre-defined paths and collects their `.eh_frame` sections. The `monitor` adheres to the same search rules as the linker [31]. Specifically, we first utilize the directories specified in the *DT_RPATH* dynamic section attribute of the binary, if present. The `rpath` of a binary or shared object is an optional entry with the *DT_RPATH* attribute in the `.dynamic` section of ELF binaries or SOs. It can be stored there at link time by the linker. If the *DT_RPATH* attribute is not found, we proceed to search for the shared objects using the default paths, such as /lib and /usr/lib. This approach ensures that we locate the necessary shared libraries required for proper execution of the program. A program can also load shared library objects at runtime by making system calls such as `dlopen`. Since the library that the program loads via `dlopen` does not exist in the `.dynamic` section, the `monitor` identifies the virtual memory area (VMA) corresponding to the shared library to locate its load address to retrieve its `.eh_frame`.

During call site checking, the `monitor` could identify that the call target of the decode

doi:10.6342/NTU202402925

call instruction is located within the program's `.plt` section. If so, the `monitor` resolves the callee's symbol to derive its actual address.

Call site checking may encounter issues due to the presence of the `b` instruction. For example, if function A branches to function B using the `b` instruction, and then function B calls function C using the `bl` instruction, the stack unwinding process in function C finds the return address pointing back to function A. In such cases, when performing call site checking in function A, the target of the `b` instruction is function B. Consequently, call site checking fails in this scenario. To address this issue, if call site checking fails, we recursively search for any `b` instructions present in the function that the return address points to in the current function . This recursive approach allows us to track down the correct call site and resolve the issue accordingly.

5.2 Forward-edge CFI Protection.

We modified Clang and LLVM's Aarch64 backend. In addition, we added an optimization pass to support our PA-based protection. We modified the `CodeGen` functions for class construction and VPointer access in the LLVM front-end. The LLVM pass analyzes the LLVM intermediate representation (IR) to identify the pointer usages discussed in Section 4.1.2. The pass also gathers essential information (i.e., function signature) to generate the PA modifiers and initializes statically allocated variables (e.g., global variables). We extended LLVM's backend to emit PA-specific instructions.

doi:10.6342/NTU202402925

5.2.1 PA modifier.

Our implementation extracts function signatures for signing function pointers in the LLVM passes via the `FunctionType` class in LLVM. This class represents the function type at the IR level. We utilize the LLVM's `TypeID` class to convert the function's signature into a string, then use the SHA3 hash function to transform the string into a 64-bit constant value. As for C++ VPointers, we modified functions that emit IR for class construction and VPointer access in the LLVM front-end. In these functions, we use the address of the VPointer as a PA modifier, as mentioned in Section 4.1.1.

5.2.2 Function Pointer Signing/Authentication.

In our work, we introduced new PA-specific LLVM intrinsics for each PA-related instruction, which pass a pointer value and a PA modifier to the backend. The information is transferred to the emitted LLVM Machine Intermediate Representation (MIR) and used when generating Arm PA instructions.

We insert instrumentation just before IR instructions that store a function address to a pointer. Subsequent accesses, such as load or store, do not authenticate the signed function pointers; instead, they are authenticated before being used in an indirect function call. Additionally, we insert instrumentation before function calls that include function addresses as their arguments. This instrumentation collects essential information such as the pointer value and the function signature. The function signature is transformed into a 64-bit constant as mentioned in the Section 5.2.1. We then insert an intrinsic call to `pacia` with this information as arguments.

doi:10.6342/NTU202402925

On the other hand, we abstain from recompiling libraries such as libc.so. Thereby, if a signed pointer is carried as argument by library functions like `pthread_create` and `sigaction` and then use it, the library can not handle the signed pointer and the program will terminate abruptly. Thus, we need to authenticate the signed pointer and remove the PAC before calling a library function. To this end, we utilize the `TargetLibraryInfo` class in LLVM to discern between user-defined function calls and library function calls. The instrumentation collects essential information such as the pointer value and the function signature, and insert an intrinsic call to `autia` with this information as arguments.

We iterate through all global variables and check if they have initializers. If an initializer is present, we determine whether its type is a function pointer. Additionally, if the initializer is an aggregate, such as a structure, we recursively examine each element to identify any function pointers. To sign these global function pointers, we introduce a new IR function, __pac_sign_globals, which is responsible for signing statically initialized global function pointers. In this function, we load the pointers and insert intrinsic calls to `pacia` for them. After defining this function, we append it to the llvm_global_ctors array. The functions referenced within this array are invoked before the `main` function execution, thereby ensuring that statically initialized global function pointers are signed beforehand.

As for function pointer authentication, we insert instrumentation just before IR instructions that used a function pointer, such as an indirect call instruction. We collect the same information as in pointer signing. Then, we insert an intrinsic call to `blraa` or `braa`, depending on the IR instruction, with this information as arguments. This approach ensures that we replace the `blr` (Branch with Link to Register) / `br` (Branch to Register) instructions in the program used against signed pointers with `blraa` (Branch with Link to Register, with pointer authentication, using a modifier and the A-key) / `braa` (Branch

doi:10.6342/NTU202402925

to Register, with pointer authentication, using a modifier and the A-key) that performs authentication and branching on function pointers with the same PA modifier.

In Listing 4, we show the actual assembly code to demonstrate the result of our instrumentation. On the top, the pointer value is loaded into a register. The 64-bit modifier is represented by four `mov` and `movk` instructions. Then, we sign the pointer value using the `pacia` instruction and store it back to the pointer. On the bottom, the pointer value is loaded into a register, and the modifier is retrieved in the same way as in pointer signing. After that, we authenticate the pointer value and branch using `blraa` instruction.

```
adrp    x8, 0 <__abi_tag-0x254>
add     x8, x8, #0x868
mov     x9, #0x47a
movk    x9, #0xb53b, lsl #16
movk    x9, #0x9224, lsl #32
movk    x9, #0xf9c, lsl #48
pacia   x8, x9
stur    x8, [x29, #-16]
...
...
ldr     x8, [x29 #-16]
mov     x9, #0x47a
movk    x9, #0xb53b, lsl #16
movk    x9, #0x9224, lsl #32
movk    x9, #0xf9c, lsl #48
blraa   x8, x9
```

Listing 4: Local function pointer

5.2.3 C++ VPointer Signing/Authentication

To sign a VPointer, we insert instrumentation before the IR instructions that store a VPointer into a class object. Additionally, we insert instrumentation immediately after the IR instructions that load a VPointer into a register for pointer authentication. The instrumentation collects essential information, such as the pointer value and the location

doi:10.6342/NTU202402925

of the pointer. We then insert an intrinsic call to `pacda` / `autda` with this information as arguments.

To implement this, we modify the `CodeGenFunction::GetVTablePtr` function in `clang/lib/CodeGen/CGClass.cpp`, which is invoked whenever VPointer is accessed. This modification ensures that pointer authentication is incorporated into every access of VPointers. Similarly, we modify the `CodeGenFunction::InitializeVTablePointer` function in the same file, which is used in class constructors to initialize the VPointer. This approach guarantees the inclusion of pointer signing during the class object initialization.

As mentioned in Section 4.1.2, VTables are typically located in a read-only memory region. Therefore, we refrain from inserting instrumentation before the virtual function call during virtual function dispatch.

In Listing 5, we present the assembly code to exemplify the results of the instrumentation for C++ VPointers. At the top, before storing a VTable into a VPointer, we first obtain the location of the VPointer within an object. Subsequently, we sign the VPointer using `pacda`, with the location serving as the modifier. At the bottom, during the retrieval of a VTable in the process of virtual function dispatch, we authenticate the VPointer using `autda`, with the location serving as the modifier. Notably, the `blr` instruction is not replace with `blraa` instruction.

doi:10.6342/NTU202402925

```
ldr      x9, [sp, #8]       ; load the location of the VPointer.
adrp     x8, 11000 <__FRAME_END__+0x10048>
add      x8, x8, #0xd48     ; load the VTable.
pacda    x8, x9             ; sign the VTable.
str      x8, [x9]           ; store it to VPointer.
...
...
ldur     x0, [x29, #-16]    ; load the location of the VPointer.
ldr      x8, [x0]           ; load signed VPointer.
autda    x8, x0             ; authenticate it.
ldr      x8, [x8]
blr      x8
```

Listing 5: C++ VPointer

doi:10.6342/NTU202402925

Chapter 6 Evaluation

6.1 Performance Evaluation

6.1.1 Experimental Setup

We conducted all benchmarks on an Apple Mac Mini M1 [8, 9], which features an ARMv8.3 architecture with ARM PA instructions. The system is equipped with 8GB of DRAM, 4 high-performance cores, and 4 high-efficiency cores. To test our loadable kernel module, we replaced macOS with Asahi Linux. For fair comparison, we ran all benchmarks without enabling any optimizations.

6.1.2 Benchmarks

We conducted experiments on Lighttpd [2], NGINX [6], and SQLite [3], three widely deployed real-world applications. These applications were chosen for their prevalence and susceptibility to security breaches, as well as their intensive I/O operations and heavy reliance on system calls. This makes them ideal candidates for us. Additionally, we evaluated the performance of our C++ VPointer protection mechanism using the SPEC-CPU2017 [13] benchmark suite, which includes benchmarks written in both pure C++ and a combination of C and C++. By using a diverse range of benchmarks from SPEC-

doi:10.6342/NTU202402925

Table 6.1: Benchmark numbers for Lighttpd, NGINX, and SQLite.

UW: Unwinder PA: Pointer Authentication

Appilcation	Unprotected	UW	PA	UW+PA
Lighttpd (Reqs/sec)	14502.71	14436.86 (0.45%)	14496.96 (0.04%)	14404.75 (0.68%)
NGINX (Reqs/sec)	13451.34	13406.41 (0.33%)	13445.628 (0.04%)	13390.73 (0.45%)
SQLite (micros/op) (write operation)	13.48	51.76 (73.96%)	13.66 (1.32%)	51.82 (73.98%)
SQLite (micros/op) (read operation)	3.75	8.95 (58.10%)	3.78 (0.79%)	9.00 (58.33%)

We access Lighttpd's and NGINX's request throughput in the number of requests per second (Reqs/sec), while SQLite is evaluated with sqlite-bench, which measures the number of microseconds per operation (micros/op).

CPU2017, we aim to comprehensively assess the effectiveness of our protection mechanism across various workloads and programming paradigms.

6.2 Application Performance

6.2.1 Lighttpd

To evaluate Lighttpd, we employed wrk [48], a renowned HTTP benchmarking tool that measures throughput by sending concurrent HTTP requests to a web server. In our setup, the wrk client operated on a separate machine within the same local network as the Lighttpd web server. The configuration of Lighttpd was set to handle a maximum of 512 connections with one worker threads. During the evaluation, we measured throughput over a duration of 20 seconds, specifically generating HTTP requests targeting a static webpage size of 6,227 bytes.

The performance breakdown is presented in Table 6.1. With the Unwinder (UW) and PA components enabled, the performance overhead is approximately 0.45% and 0.68%, respectively.

doi:10.6342/NTU202402925

6.2.2 NGINX

We use the same benchmarking tool and setup for NGINX. Our NGINX configuration is set to handle a maximum of 512 connections per processor with 8 worker threads. During the evaluation, we measure throughput over a 20 second duration. To simulate real-world scenarios, wrk spawns the same number of threads as NGINX's configured worker count, with each wrk thread generating HTTP requests for a 6,227 byte static webpage.

The runtime overhead for NGINX minimally increased with all components enabled. When full protection (with all components: PA, Unwinder) was applied, the overhead never exceeded a 0.45% degradation compared to the baseline vanilla NGINX from Table 6.1.

6.2.3 SQLite

SQLite [3] stands as a widely deployed, transactional SQL database engine. To assess the performance throughput of SQLite, we employ the sqlite-bench [5] which is a C version of SQLite benchmark in Google's levelDB [22]. We utilize the sqlite-bench to simulate read and write operations for large data warehouse transactions. We select two benchmarks in the sqlite-bench, one is `fillrandom` writing N values in random key order in async mode, and another is `readrandom` reading N times in random order. We ran these benchmarks with the default settings, utilizing a value of N equal to 1,000,000. We measured sqlite-bench's performance in terms of the number of microseconds per operation.

In Table 6.1, we can observe that the overhead for SQLite stands out as significantly higher compared to other benchmarks on UW. This is primarily because sqlite-bench ex-

doi:10.6342/NTU202402925

Table 6.2: System call usage

Application	SQLite	NGINX	Lighttpd
execve	0	0	0
execveat	0	0	0
clone	0	8	1
mprotect	5	4	6
mmap	11	11	12
mremap	0	0	0
chmod	0	0	0
setuid	0	0	0
setgid	0	0	0
setreuid	0	0	0
socket	0	1	1
bind	0	1	1
connect	0	0	0
listen	0	2	1
accept	0	0	0
accept4	0	275,304	517
openat	17	269,563	10
read	1,634,533	5	589,310
write	6,302,641	269,551	1
readv	0	0	0
writev	0	269,551	294,485
sendfile	0	269,551	0
recvfrom	0	269,748	432
Total number of system calls	7,937,201	1,623,300	884,779

doi:10.6342/NTU202402925

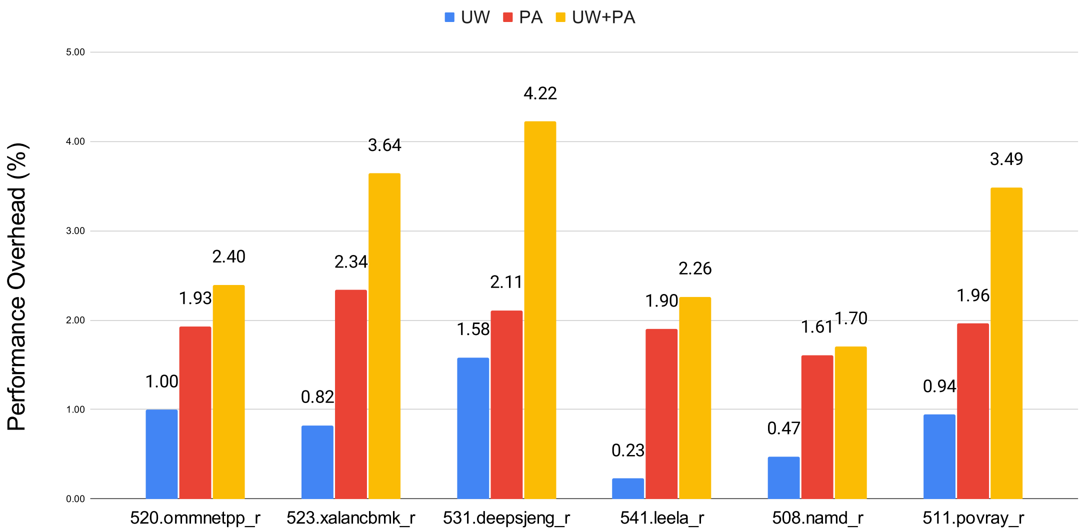

Figure 6.1: Performance overhead for SPEC CPU2017.

ecutes more write/read operations stantially (see Table 6.2).

6.2.4 SPECCPU2017

To test our C++ VPointer protection mechanism, we select 5 pure C++ and 1 both C and C++ benchmarks from SPECCPU2017. The performance overhead for these benchmarks is depicted in Figure 6.1. When all protections are enabled, the average overhead is 2.95% compared to the unprotected baseline benchmarks. From Figure 6.1, we observe that the overhead of PA is greater than that of UW. This disparity arises because authentication is inserted in all accesses to VPointers, and these benchmarks feature heavy virtual function calls. Consequently, the overhead incurred by PA is notably higher compared to UW.

doi:10.6342/NTU202402925

Chapter 7 Limitation and Discussion

In this chapter, we will discuss the limitations of the unwinder. Specifically, we will examine the protection of the unwinder in scenarios with incomplete and complete unwinding information, respectively.

7.1 The Limitation of the Unwinder.

In Section 4.1.2, we have demonstrated that call site checking only works on direct call instructions. If an indirect call instruction is encountered during unwinding, we cannot verify if the target of the call instruction is the start of the function. Therefore, we cannot determine if a return address is safe and has not been tampered with by attackers. Attackers can exploit this weakness in call site checking to counterfeit a call path that does not exist in the program. For example, attackers can effectively bypass the entire protection of the unwinder by using a special gadget. This gadget must be located in the `main` function and have an indirect call instruction immediately above it, allowing the attackers to circumvent the unwinder's security measures. This special gadget satisfies our observations: (1) unwinding reaches the `main` function, and (2) call site checking succeeds due to the indirect call instruction.

On the other hand, if no such gadget is found, attackers can still exploit the weakness

doi:10.6342/NTU202402925

in call site checking. They can tamper with return addresses using gadgets with an indirect call instruction immediately preceding them. By doing so, attackers can counterfeit or mimic a valid call path, bypassing the unwinder's protection mechanisms.

7.1.1 Complete Unwinding Information.

This section considers the scenario where attackers have complete unwinding information. As detailed in Section 2.1, calculating the Canonical Frame Address (CFA) for the stack frame being unwound is essential to retrieve the current return address. Consequently, attackers with full unwinding information can easily locate the return address using the CFA.

Suppose a function contains vulnerabilities, such as a stack buffer overflow. In that case, attackers can perform an ROP attack and leverage the CFA for the vulnerable function's stack frame to bypass our protection if they possess complete unwinding information. During an attack, they can overwrite the return address, located using the CFA, to bypass the unwinder's protection mechanism. Specifically, attackers can counterfeit or mimic a valid call path or exploit the weakness of call site checking using their knowledge of CFAs to bypass the unwinder.

7.1.2 Incomplete Unwinding Information.

We assume that attackers cannot acquire the complete unwinding information because they cannot access the binary running on the remote server. Even if attackers know the version of the binary, they still cannot obtain the complete unwinding information. This is because the specific dependencies used by the binary are unknown to them. With-

doi:10.6342/NTU202402925

out the knowledge of the CFA, attackers cannot easily overwrite return addresses to counterfeit or mimic the entire call path.

7.2 Attacks on PAC

PAC can be bypassed if an attacker with read/write access can coerce the program into executing a signing gadget [41]. Signing gadgets are sequences of instructions that can be exploited to sign arbitrary pointers. For instance, if an attacker can trigger the execution of a function that reads a pointer from memory, adds a PAC, and writes it back, they can use this function as a signing oracle to forge PACs for arbitrary pointers.

Our current implementation focuses on ensuring the control flow integrity of system call usages; the unwinder is triggered when a monitored system call is invoked. Since exploiting signing gadgets is unrelated to the integrity of system call usage, we cannot detect an attack if an attacker hijacks the control flow to execute a signing gadget.

To address this issue, we can extend PAC to protect return addresses. We have already provided protections for function pointers and C++ VPointers. By integrating our work with PAC for protecting return addresses, any corrupted return addresses or call targets will trigger authentication failures, ensuring that such attacks are detected before the program's control flow can be hijacked.

doi:10.6342/NTU202402925

Chapter 8 Security Evaluation

8.1 Security Analysis

We conducted case studies on various attacks, as detailed in Table 8.1. These include Return-Oriented Programming (ROP) attacks, real-world vulnerabilities such as CVE-2012-0809, CVE-2013-2028, CVE-2015-8617, CVE-2016-10190, and CVE-2016-10191, as well as advanced attack techniques proposed in [45], [18], and [47].

8.1.1 ROP

ROP fundamentally relies on placing gadgets on the stack and overwriting at least one return address to execute an attack. However, these malicious actions corrupt the stack, which disrupts the stack unwinding process and causes it to fail.

8.1.2 VPointer Hijacking

We evaluated our VPointer protection mechanism using five synthesized attacks in C++ to demonstrate our ability to defend against VPointer hijacking attacks and COOP attacks. For this evaluation, we used the CFIXX C++ test suite [34] by Burow et al. [14], which contains four VPointer hijacking exploits and one COOP exploit.

doi:10.6342/NTU202402925

Table 8.1: Security Analysis against Attacks

✓: protected by component; ✗: bypass the component.
UW: Unwinding, PA: Pointer Authentication

Attack Scenario	**UW**	**PA**
ROP	✓	✗
VPointer Hijacking	✗	✓
Direct System call Manipulation		
Newton CsCFI [47]	✗	✓
CVE-2016-10190 ffmpeg [38]	✓	✓
CVE-2016-10191 ffmpeg [39]	✓	✓
CVE-2013-2028 nginx [35]	✓	✓
CVE-2012-0809 sudo [36]	✓	✓
CVE-2015-8617 php [37]	✓	✓
Indirect System Call Manipulation		
Newton CPI [47]	✗	✗
Control Jujutsu Nginx [18]	✗	✓
AOCR Nginx 2 [45]	✗	✓
AOCR Httpd [45]	✗	✓

doi:10.6342/NTU202402925

We detected all exploits by authenticating the VPointer during virtual function dispatch. In VPointer hijacking exploits, attackers either (1) overwrite a class object with another class object or (2) overwrite a VPointer. In the first case, we detect the attack because the two objects are located in different memory locations. In the second case, we detect this because the fake VPointer lacks a PAC, causing the authentication to fail. In the COOP attack, a fake object is crafted without invoking the constructor (e.g., directly using `malloc()`), and the VPointer of the fake object is utilized. We detect this because the VPointer was never initialized and lacks a PAC, thus causing the authentication to fail.

8.1.3 Direct System Call Manipulation

We present attack scenarios where attackers exploit memory corruption vulnerabilities and discuss how to defend against these attacks. Newton CsCFI [47] exploits a bug in Nginx to hijack a function pointer and use `mprotect` to make libc's memory mapping read-write-executable (RWX). This enables attackers to inject shellcode into libc, leading to arbitrary code execution. We counter this attack with function pointer protection. Additionally, vulnerabilities such as CVE-2016-10190, CVE-2016-10191, CVE-2013-2028, and CVE-2019-3822 result in buffer overflows, while CVE-2015-8617 and CVE-2012-0809 involve format string vulnerabilities that allow arbitrary writes. These vulnerabilities can potentially tamper with data in memory. Our protective measures effectively guard against attacks targeting return addresses and function pointers, thereby mitigating the risk posed by these vulnerabilities.

doi:10.6342/NTU202402925

8.1.4 Indirect System Call Manipulation

AOCR Nginx Attack 2 [45] exploits a stack buffer overflow to overwrite a function pointer, redirecting the execution flow to the `ngx_master_process_cycle` function in Nginx. AOCR then manipulates a conditional variable within this function to bypass an 'if' statement and invoke `execve`. The attack uses malicious data provided by remote attackers as arguments to `execve`. We prevent such attacks by ensuring function pointer integrity. The AOCR httpd [45] and Control Jujutsu Nginx [18] attacks each target a function pointer in Apache and Nginx, respectively, overwriting the pointer to redirect execution to another function that calls `execve`. We use function pointer integrity to protect against such attacks. However, the Newton CPI Nginx attack [47] bypasses this protection by manipulating non-pointer values. Specifically, it corrupts a variable indexing a structure array to cause an out-of-bounds access, redirecting a function pointer to `mprotect`.

doi:10.6342/NTU202402925

Chapter 9 Related Work

9.1 Related Work

9.1.1 Debloating and system call filtering.

Debloating-based approaches [7, 42, 43] reduce the program's attack surface by eliminating unused code. System call filtering methods create a legal set of system calls that a program can execute, using seccomp-BPF [24] filters to block any outside this set. Some [16, 17] automate the generation of these filter sets, while [20] splits application execution into phases, creating distinct filter sets for each phase. However, these whitelist-based methods cannot prevent attackers from abusing sensitive system calls within the legal set. In contrast, we ensure the execution flow is not hijacked when a process invokes sensitive system calls.

9.1.2 Runtime System Call Protection.

Previous work [15, 25, 27] checks a system call control flow graph (CFG) to ensure the integrity of system call usage. Unlike our work, [15, 27] do not secure system call arguments so that they could be corrupted by attackers. Further, none of the works supports Arm. The CFG that those works rely on could be unavailable in program bina-

doi:10.6342/NTU202402925

ries. Program recompilation required to generate the CFGs may not be feasible since the source code and library are likely proprietary and maintained by third-party vendors in Arm ecosystems, limiting adoption in practice.

Bastion [25] hooks sensitive system calls to protect their usage, relying on Linux's ptrace to monitor and intercept these calls. When an application makes a sensitive system call, Bastion performs a context switch to the monitoring process to validate the system call's usage, and another context switch is required to return control to the application. These context switches introduce significant overhead, particularly in network applications that frequently invoke system calls. For example, Bastion causes a 95% drop in NGINX's throughput, while our approach incurs a modest 0.45% drop.

To protect system calls, Bastion determines the legitimate call type (direct or indirect function call) of system calls during compilation. At runtime, it checks whether a system call is invoked through a legal means. However, Bastion focuses only on the call type of the system call itself, not on the call type of functions in the call chain that invoke the system call. For example, AOCR Nginx Attack 2 [45] demonstrates compromising Nginx by overwriting a function pointer to a function that invokes a sensitive system call. Although it is illegal to invoke the caller function through an indirect function call, Bastion would not detect the attack because it only checks the legality of the system call's invocation, not the call chain leading to it. In contrast, we enhance security by protecting function pointer integrity through Pointer Authentication (PA) and argument integrity. These measures can detect all attempts to hijack a function pointer, offering a more comprehensive defense against such attacks.

doi:10.6342/NTU202402925

9.1.3 Pointer Integrity Protection.

Previous work enforces pointer integrity through various approaches, such as SafeStack [26], CCFI [33] and CFIXX [14]. Code Pointer Integrity protects access to code pointers, as well as data pointers that may reference code pointers, by storing them in a separate area of memory known as the SafeStack. The SafeStack itself must be protected from unauthorized access to maintain its security. Stronger protection mechanisms for the SafeStack, such as hardware-enforced isolation or software-based isolation, impose an average performance overhead of 8.4% and 13.8%, respectively, as measured in the SPECCPU benchmarks. However, SafeStack with hardware-enforced isolation incurs significant overheads on C++ benchmarks, such as omnetpp($\approx$ 44%), xalanbmk($\approx$ 37%) and povray($\approx$ 42%). In contrast, our work shows a lower average overhead of approximately 2.9% on these benchmarks while providing protections for function pointers, C++ VPointers.

CCFI employs Message Authentication Codes (MACs) to safeguard return addresses, function pointers, and VPointers. Conceptually, the use of MACs is analogous to Pointer Authentication (PA). However, CCFI does not leverage hardware-accelerated PA instructions, resulting in a significant performance overhead. Specifically, CCFI incurs an average overhead of 52% when evaluated across the SPEC CPU2006 benchmarks. In our work, we leverage the benefits of PA to provide the same protection, but with much lower overhead (2.95% on average).

CFIXX accomplishes object type integrity (OTI) by protecting the VPointers, which fundamentally requires only legitimate writing to be allowed on these pointers. During object construction, it records the correct VPointer in a metadata table. During virtual

doi:10.6342/NTU202402925

function dispatch, it retrieves and uses this correct VPointer from the metadata table. This approach ensures that the correct VPointers are used, preventing the use of potentially attacker-corrupted VPointers in the object. The scheme imposes approximately a 13% slowdown in the worst case for C++ benchmarks in SPECCPU. In contrast, we incur only a 4.22% slowdown in the worst case for C++ benchmarks in SPECCPU from Figure 6.1.

9.1.4 PAC Defense Approaches.

Previous work also leverages PA to protect pointer integrity, such as PARTS [28] and PACTight [23].

PARTS safeguards data pointers, code pointers, and return addresses using PA. To be specific, it employs LLVM's ElementType to generate a type ID, which acts as a PA modifier to protect these pointers and return addresses. However, this method does not cover the protection of C++ VPointers. In this thesis, we further extend protection to C++ VPointers.

On the other hand, PACTight extends protection not only to data pointers, code pointers, and return addresses but also to C++ VPointers. it adopts a different approach by combining the pointer's location with a 64-bit random tag as the PA modifier. However, this method incurs overhead due to the need to store and manage random tags for each protected pointer. This overhead is particularly significant in programs with frequent virtual function calls, such as omnetpp and xalancbmk from the SPECCPU benchmark suite, resulting in approximately 9% and 6% overhead, respectively. From Figure 6.1, with full protections incurs a lower runtime overhead (2.40% on omnetpp and 3.64% on xalancbmk) compared to PACTight.

doi:10.6342/NTU202402925

9.1.5 Unwinding based Approaches.

Similar to our work, SLICK [19] uses stack unwinding to detect ROP attacks and safeguard backward-edge CFI. However, unlike the callsite checking employed by us, SLICK identifies stack corruptions by verifying if the runtime stack layout aligns with the anticipated layout derived from statically identified stack operations of the program. We plan to explore integrating SLICK's approach into our kernel module as future work.

doi:10.6342/NTU202402925

Chapter 10 Conclusions

This thesis is based on the observation that, regardless of an attack's complexity or ultimate goal, most attack must utilize system calls to achieve their objectives. Therefore, we ensure a secure property: **the control flow integrity of system call invocations**. In this thesis, we implement a prototype and evaluate our work to access the performance impact on system call-intensive applications, ultimately demonstrating low runtime overhead. This indicates that our approach is practical for disrupting attackers' ability to achieve their malicious objectives through system calls.

doi:10.6342/NTU202402925

References

[1] Dwarf debugging information format, version 4. https://dwarfstd.org/doc/DWARF4.pdf.

[2] Lighttpd web server. https://www.lighttpd.net/.

[3] Sqlite. https://www.sqlite.org/index.html.

[4] Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors, Nov. 2018. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances.

[5] A sqlite3 benchmark tool, 2018. https://github.com/ukontainer/sqlite-bench.

[6] Nginx web server, 2022. https://nginx.org.

[7] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis. Nibbler: debloating binary shared libraries. In Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19, page 70–83, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Apple. Apple mac mini m1, 2020. https://www.apple.com/shop/buy-mac/mac-mini/applem1-chip-with-8-core-cpu-and-8-core-gpu-256gb.

doi:10.6342/NTU202402925

[9] Apple. Apple unleashes m1, 2020. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[10] Apple Inc. Apple platform security, May 2022. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf.

[11] Arm Developer. Execute never, 2014. https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Memory-attributes/Execute-Never?lang=en.

[12] R. Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. IACR Transactions on Symmetric Cryptology, pages 4–44, 2017.

[13] J. Bucek, K.-D. Lange, and J. v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pages 41–42, 2018.

[14] N. Burow, D. McKee, S. A. Carr, and M. Payer. Cfixx: Object type integrity for c++ virtual dispatch. In Symposium on Network and Distributed System Security (NDSS), 2018.

[15] C. Canella, S. Dorn, D. Gruss, and M. Schwarz. Sfip: Coarse-grained syscall-flow-integrity protection in modern systems. arXiv preprint arXiv:2202.13716, 2022.

[16] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating seccomp filter generation for linux applications. In Proceedings of the 2021 on Cloud Computing Security Workshop, pages 139–151, 2021.

doi:10.6342/NTU202402925

[17] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), pages 459–474, 2020.

[18] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[19] Y. Fu, J. Rhee, Z. Lin, Z. Li, H. Zhang, and G. Jiang. Detecting stack layout corruptions with robust stack unwinding. In Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19, pages 71–94. Springer, 2016.

[20] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. Temporal system call specialization for attack surface reduction. In 29th USENIX Security Symposium (USENIX Security 20), pages 1749–1766. USENIX Association, Aug. 2020.

[21] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In 2014 IEEE Symposium on Security and Privacy, pages 575–589. IEEE, 2014.

[22] Google. A fast key-value storage library, 2011. https://github.com/google/leveldb.

[23] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min. Tightly seal your sensitive pointers with {PACTight}. In 31st USENIX Security Symposium (USENIX Security 22), pages 3717–3734, 2022.

doi:10.6342/NTU202402925

[24] Jake Edge. A library for seccomp filters. https://lwn.net/Articles/494252/.

[25] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min. Protect the system call, protect (most of) the world with bastion. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, pages 528–541, 2023.

[26] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In The Continuing Arms Race: Code-Reuse Attacks and Defenses, pages 81–116. 2018.

[27] L. C. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings 7, pages 1–20. Springer, 2004.

[28] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In 28th USENIX Security Symposium (USENIX Security 19), pages 177–194, 2019.

[29] Linux Foundation. Dynamic section. https://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/dynamicsection.html.

[30] Linux Foundation. Section header. https://refspecs.linuxbase.org/elf/gabi4+/ch4.sheader.html.

[31] Linux manual page. ld.so. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[32] Mark Rutland. Armv8.3 pointer authentication, September 14, 2017.

doi:10.6342/NTU202402925

https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf.

[33] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Nathan Burow. Cfixx c++ test suite, 2018. https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite.

[35] https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[36] https://nvd.nist.gov/vuln/detail/CVE-2012-0809.

[37] https://nvd.nist.gov/vuln/detail/CVE-2015-8617.

[38] https://nvd.nist.gov/vuln/detail/CVE-2016-10190.

[39] https://nvd.nist.gov/vuln/detail/CVE-2016-10191.

[40] PaX. Address space layout randomization, 2003. https://pax.grsecurity.net/docs/aslr.txt.

[41] Project Zero. Examining pointer authentication on the iphone xs, Feb 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[42] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. {RAZOR}: A framework for post-deployment software debloating. In 28th USENIX security symposium (USENIX Security 19), pages 1733–1750, 2019.

doi:10.6342/NTU202402925

[43] A. Quach, A. Prakash, and L. Yan. Debloating software through Piece-Wise compilation and loading. In 27th USENIX Security Symposium (USENIX Security 18), pages 869–886, Baltimore, MD, Aug. 2018. USENIX Association.

[44] Qualcomm Technologies, Inc. Pointer authentication on armv8.3, January 2017. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[45] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. Address-oblivious code reuse: On the effectiveness of leakage-resilient diversity. Network and Distributed System Security Symposium, 2017.

[46] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In 2015 IEEE Symposium on Security and Privacy, pages 745–762. IEEE, 2015.

[47] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. ACM SIGSAC Conference on Computer and Communications Security, 2017.

[48] Will Glozer. a http benchmarking tool, 2019. https://github.com/wg/wrk.

[49] C. Williams. Microsoft: Can't wait for ARM to power MOST of our cloud data centers! Take that, Intel! Ha! Ha! The Register, Mar. 2017. https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup.

doi:10.6342/NTU202402925