Strengthening Application Security through Integrity Protection of System Call Usage

Chih-Kai Hsu
National Taiwan University
r11944008@csie.ntu.edu.tw

Yung-Chi Tseng
National Taiwan University
r11922168@csie.ntu.edu.tw

Shih-Wei Li
National Taiwan University
shihwei@csie.ntu.edu.tw

Yi-Hui Lin
Delta Research Center
YIHUI.YH.LIN@deltaww.com

Wei-Cheng Tian
Delta Research Center
WC.TIAN@deltaww.com

Lap Chung Lam
Delta Research Center
lapchung.lam@gmail.com

Yu-Chen Liu
National Taiwan University
University of Southern California
yliu6359@usc.edu

Abstract

Attackers who exploit program vulnerabilities leverage system calls provided by the OS kernel to execute malicious actions, posing significant security risks. This work introduced Scynteg, a new framework for protecting system call usage in modern applications. Scynteg enforces control flow integrity of system call invocations and system call argument integrity through a combination of a secure kernel module and an LLVM-based compiler. We show that Scynteg effectively protects C/C++ programs against control-flow hijacking and non-control-data attacks on system call arguments. Our prototyped implementation for Linux on Arm leverages hardware security extensions to effectively protect applications while incurring a modest performance overhead.

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 executing harmful payloads.

Various approaches have been proposed to address the system call security issues. An intuitive approach is to reduce the program's attack surface. Some work leverages debloating [3, 45, 46] to avoid running unused code that could use sensitive system calls. Others [12, 18, 22] profile programs' behaviors and filter out unused system calls. However, these approaches cannot prevent the execution of sensitive system calls (e.g., execve) essential to the programs' functionalities. Other past work has addressed these limitations by enforcing runtime integrity of system call invocations [11, 27, 28] against a system call control flow graph (CFG) during the program's execution. These solutions rely on complex software approaches that complicate protection efforts, potentially resulting in high-performance overhead.

In this paper, we proposed Scynteg, a new framework for protecting system call usage for C/C++ programs on Linux-based platforms. Scynteg ensures two properties: ***(P1) the control flow integrity of system call invocations, and (P2) the integrity of system call arguments***. Scynteg employs a runtime and static time approach that consists of a loadable kernel module and an LLVM-based compiler to collectively enforce the intended system call security properties.

To enforce property (P1), Scynteg's kernel module implements a secure `monitor` that protects the backward-edge control flow integrity (CFI) of system call invocations using direct function calls. The monitor hooks Linux's system call table and incorporates a stack unwinding-based approach. When a task invokes a sensitive system call, the monitor unwinds the caller's call stacks. It validates whether the program made the call via a legal control-flow function call path. Scynteg leverages the program's DWARF [15] debug information readily available from the program's .eh_frame section to perform call stack unwinding. The monitor requires no changes to Linux and can be dynamically loaded to various Linux-based systems with ease. In addition, Scynteg introduces an LLVM pass to ensure forward-edge CFI. Scynteg leverages hardware security extensions to efficiently protect the integrity of function pointer usage and virtual method invocations in C++ programs.

To further secure system call usage, Scynteg enforces property (P2) to prevent attackers from crafting system call arguments and using the crafted arguments against system calls on legitimate function call paths. Scynteg employs taint analysis to track a program's modifications to the system call arguments. It seals the updated argument values in a protected shadow memory area. Scynteg approves the system call invocation only if the arguments match the respective sealed values.

Given their increasing adoption, we built a Scynteg prototype to protect system call usage for applications running on Arm-based systems. The prototype supports widely used network-facing server applications on stock Linux distributions running on real Arm hardware. Scynteg protects the control flow integrity of system call invocations for these programs while securing the integrity of system call arguments. Scynteg utilizes Arm's pointer authentication (PA) [33, 47] feature to efficiently secure forward-edge CFI. Our security analysis shows that Scynteg effectively protects against vulnerabilities that exploit control-flow hijacking approaches, including Return-Oriented Programming (ROP), Signal Return Oriented Programming (SROP), and Catch Handler Oriented Programming

(CHOP), to result in malicious system call invocations. Scynteg also effectively mitigates data-oriented attacks that aim to corrupt system call arguments. The Bulwark prototype imposes a modest overhead on protected applications, underscoring its efficiency and practical viability in enhancing system call security without significantly compromising performance.

2 Background

2.1 Debug Information: 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 toolchain in the DWARF[15] 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 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.

```
1
2  00000000000408a4 <ngx_alloc>:
3
4  408a4:   a9bd7bfd    stp     x29, x30, [sp, -48]!
5  408a8:   910003fd    mov     x29, sp
6  408ac:   a90153f3    stp     x19, x20, [sp, 16]
7  408b0:   f90013f5    str     x21, [sp, 32]
8  408b4:   aa0003f5    mov     x21, x0
   408b8:   aa0103f4    mov     x20, x1
```

Listing 1: assembly code of `ngx_alloc`

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

Listing 2: FDE of `ngx_alloc`

2.2 Catch Handler Oriented Programming (CHOP)

Modern programming languages like C++ support exception handling, allowing programs to throw exceptions and define exception handlers (EHs) to catch them. When an exception is raised within a landing pad (e.g., a try block), control flow transfers to the associated EH. Exception handling involves two phases: *Search Phase*: An unwinder examines call frames to locate a suitable EH without unwinding the stack. This phase ends when an EH is found, or the stack is exhausted. *Cleanup Phase*: The unwinder unwinds stack frames from the exception's origin to the found EH, invoking cleanup handlers (e.g., destructors) for each frame. Control then transfers to the EH, which typically terminates the program in languages implementing termination semantics.

The Catch Handler Oriented Programming (CHOP) attack exploits the exception-handling mechanisms by corrupting the stack to replace saved return addresses and hijack control flow. The attacker then crafts attacker-controlled data used by the unwinder or handlers. Attackers can redirect execution to unintended EHs or cleanup handlers, potentially replacing a function's return address with one within a desired landing pad. This manipulation allows the execution of attacker-chosen functions.

CHOP Attack Example. Listing 3 shows a potential CHOP attack scenario using two functions from the standard C++ library: `__cxa_call_unexpected` and `xh_terminate_handler`. The former contains a catch-all handler capable of catching all exceptions. To execute a CHOP attack, an attacker could first corrupt the stack to overwrite the faulting function's return address, redirecting it to an address within the try clause of `__cxa_call_unexpected`, then replace `xh_terminate_handler` with the address of their intended malicious function. Upon exception occurrence, libstdc++ would execute the catch clause in `__cxa_call_unexpected`, subsequently invoking `__terminate`. This, in turn, would execute the attacker's intended function through the `handler()` call.

```
void __cxa_call_unexpected(void *exc_obj_in)
{
/*...*/
    xh_terminate_handler = xh->terminateHandler;
    /*...*/
    __try {
        /*...*/
    }__catch(...){
        /*...*/
        __terminate(xh_terminate_handler);
    }
    /*...*/
}

void __terminate(std::terminate_handler handler)
    throw()
{
    __try{
        handler();
        std::abort();
    }__catch(...) {
        std::abort();
    }
}
```

Listing 3: libstdc++ `__cxa_call_unexpected`

2.3 Signal Return Oriented Programming (SROP)

When a process receives a signal, the kernel stores the interrupted process' states, including its signal mask and execution context (processor flags and register values, including the stack pointer and program counter), into a signal frame in the stack. After signal handling completes, the signal handler switches to the signal trampoline, which makes the *sigreturn* system call to restore the saved states from the signal frame, such that the process continues from where it was interrupted.

The Linux kernel does not track the signals it delivers, so it cannot check whether a *sigreturn* is made after signal handling completes. Further, when restoring the signal frame from the stack, *sigreturn* does not check whether the signal frame restored is the one that it stored while receiving a signal. Attackers exploit these properties to formulate an SROP attack. To perform the SROP, the attacker creates a fake signal frame on the stack and executes a gadget to invoke *sigreturn*. This results in the frame's contents being loaded into hardware registers, including the program counter, enabling redirection of execution to an arbitrary address. Previous work [9] mentioned that attackers can find gadgets that execute the system call and return instruction (i.e., `svc 0x0; ret` on Arm64) from Linux's Virtual Dynamic Shared Object (VDSO).

2.4 Arm Pointer Authentication

Arm introduced Pointer Authentication (PA) [33, 47] 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 [8] cipher. PA provides five keys, 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 (salt), 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 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 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.

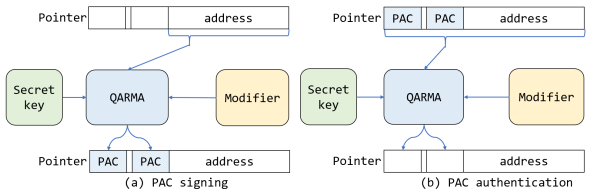

Figure 1: PAC signing and authentication

3 Threat Model and Assumptions

We aim to disrupt the attacker's ability to make system calls to perform malicious actions. Scynteg protects the sensitive system calls listed in Table 1. We assume a powerful adversary capable of reading and writing arbitrary memory by exploiting one or more memory vulnerabilities, such as heap or stack overflows, in a program. An attacker could corrupt the call stack to hijack the program's control flow. The targeted system employs Execute Never [7] (XN) and Address Space Layout Randomization [44] (ASLR) to prevent attackers from injecting or modifying code. We also assume a hardware or software shadow stack will be employed (e.g., LLVM Shadow Call Stack (SCS) [50], Guarded Control Stack (GCS) [34] on ARMv9-A). The hardware and the OS kernel are trusted. We exclude side-channel attacks from our threat model. We focus on protecting control-data attack and non-control-data attack [14] on system call arguments, and data-oriented programming [23] is out of scope.

Table 1: Protected system call set.

System call category	Available System calls
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`

4 Scynteg Design

We aim to protect an application from attacks that exploit sensitive system calls through code reuse attacks. Therefore, we aim to protect the integrity of a program's usage of sensitive system calls. To achieve the goal, Scynteg focuses on protecting two properties: (P1) the control flow integrity of system call invocations, and (P2) the integrity of system call arguments.

4.1 Control Flow Integrity of System Call Usage

Scynteg protects the control flow integrity of direct and indirect function calls. For the former, Scynteg relies on a kernel `monitor` module that employs a stack unwinding-based approach to enforce the integrity of a function call path that leads to a system call invocation. Scynteg uses hardware extensions (e.g., Arm's PA) to prevent hijacking indirect function calls.

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 valid call instruction (e.g., `bl` instruction in Arm64).

Scynteg's monitor module hooks Linux's system call table to interpose the invocation of sensitive system calls. Based on the observations, Scynteg develops a security policy that the monitor enforces at runtime. It performs algorithm 1 to validate the legality of the function call path that leads to the system call invocation. At a high level, the algorithm enforces the intended security policy. If the monitor detects a violation against the policy, it uncovers that the call stack was corrupted to invoke a sensitive system call, henceforth stops the illegal invocation.

The monitor acquires the stack layout information of a program from the program's `eh_frame` section. The information is readily available in C/C++ programs. 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 *callsite checking*. Callsite 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. The loop body first retrieves the `.eh_frame` by the current program counter (PC). If this retrieval is unsuccessful, the monitor attempts to find the VMA by PC and then gets the `.eh_frame` using VM_FILE (line 12). Next, the monitor gets the current function's entry point by the `.eh_frame` and PC (line 14). If the current function's entry point is `main` or is `__clone` (for multi-threaded programs, `__clone` is the start function of a worker thread), the monitor concludes that the call path is valid; otherwise, the monitor processes the CFI from the `.eh_frame` and restores the register according to the CFI (line 18 to 22). 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 callsite 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) 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. Scynteg protects forward-edge control flow integrity to prevent attackers from hijacking indirect function calls. Specifically, Scynteg focuses on protecting the integrity of function pointers so attackers cannot corrupt them to redirect the program's control flow. In addition, Scynteg protects virtual function usages in C++ programs. 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 VPointer for an instantiated class object that points to the object's respective VTable. In C++ programs, VTables are located in a read-only memory region. We observe that, instead of compromising VTable entries, attackers exploit vulnerabilities to corrupt Vtable pointers (VPointers) to point to a crafted VTable containing malicious pointers to hijack the program's control flow. Because of this, Scynteg focuses on protecting the integrity of VPointers.

Algorithm 1: Unwinder

```
1  Function Unwinding:
2      while true do
3          PC = REGS → PC; EhFrame = GetEhFrame(PC)
4          if EhFrame does not exist then
5              VMA = GetVMA(PC)
6              if VMA is invalid or does not exist then
7                  Abort the process
8              end
9              if VMA → VM_FILE does not exist then
10                 Abort the process
11             end
12             EhFrame = GetEhFrameFromVMA(VM_FILE)
13         end
14         EntryPoint = GetEntryPoint(EhFrame, PC)
15         if EntryPoint is main or EntryPoint is __clone then
16             return Success
17         end
18         CFI = ProcessCFI(EhFrame)
19         if CFI is invalid then
20             Abort the process
21         end
22         REGS = UpdateREGS(REGS, CFI)
23         RA = REGS → X30
24         CallInsn = DisassembleInsn(RA - 4)
25         if CallInsn is really a call instruction then
26             Callee = GetCalleeFromCallInsn(CallInsn)
27             if Callee is not the same as EntryPoint then
28                 Abort the process
29             end
30         else
31             Abort the process
32         end
33         REGS → PC = RA
34     end
```

4.2 Argument Integrity

Scynteg enforces data flow integrity of the system call arguments, function pointers, and their upward exposed variables. We denote the protected data collectively as *sensitive data* in the rest of the paper. We use an example to motivate Scynteg's protection strategy.

4.2.1 Attack Scenario. To corrupt system call arguments, an attacker could corrupt their dependent variables in the function call path or pass arbitrary values to function calls. Take Listing 4 as an example; assume that an attacker aims to craft the `flag` argument sent to the `open` system call in line 24. She could (1) overwrite the

Table 2: Argument Protection API

API name	purpose
`api_record(addr,size,isPtr)`	records shadow copy of `addr` with `size` and whether `isPtr`
`api_record_reg(i,addr)`	binds register `i` to `addr`
`api_check_reg(i)`	checks whether value in the `i`th register is consistent with where it binds with
`api_check()`	check all shadow memory

value stored in the memory of the local variable `flag`, (2) overwrite the value of the first argument passed to `bar` (line 6).

```
1  void foo() {
2      int flag = O_WRONLY;
3      //api_record(&flag, sizeof(flag), false);
4
5      //api_record_reg(0, &flag);
6      bar(flag);
7  }
8  void bar(int flag) {
9      //api_check_reg(0);
10
11     char *path = "a.txt";
12     //api_record(&path, sizeof(path), true);
13
14     int flag2 = O_CREAT;
15     //api_record(&flag2, sizeof(flag2), false);
16
17     flag |= flag2;
18     //api_record(&flag, sizeof(flag), false);
19
20     //api_record_reg(0, &path);
21     //api_record_reg(1, &flag);
22     //api_check();
23
24     open(path, flag);
25 }
```

Listing 4: code example

4.2.2 Methodology. Scynteg instruments programs to track their legitimate updates to data relevant to the system call arguments and keep the most recent data contents in a secure shadow memory. The shadow memory is a hash table. Each table entry maps the address of a data to a triple: `(data, size, isPtr)`. When the program invokes a sensitive system call, each of the arguments and all their associated upward exposed variables (i.e., variables whose definitions reach the sink variable through data-flow paths), Scynteg compares the respective value stored in *shadow memory* with the value used by the call. If all values are identical, Scynteg validates the integrity of arguments; otherwise, Scynteg identifies a corruption of the arguments and rejects the call. This is based on an intuition that if the content stored in a memory address is written by illegal means, such as buffer overflow, the instrumented API would not be executed; therefore, the content in shadow memory would not be updated, resulting in a discrepancy between shadow memory and the actual argument.

4.2.3 API. As shown in Table 2, `api_record` tracks legitimate data updates, storing values in shadow memory and optionally preserving dereferenced pointer values (if `isPtr` is true). `api_record_reg` binds a register with memory addresses in shadow memory, while `api_check_reg` validates function arguments by comparing register values with their associated memory values. This validation occurs at the start of functions, as exemplified in Listing 4, where it ensures the correct passing of `flag` in line 6. Lastly, `api_check` verifies data consistency between shadow and actual memory, deallocating shadow memory upon completion.

4.2.4 Sensitive Variable Instrumentation. For functions involving function pointers or sensitive system calls, Scynteg performs intra-procedural analysis to identify sensitive data correlated with system call arguments. When such data is passed from a caller function, it recursively traces the call chain backward to the origin of the use-def chain (e.g., where a variable is first allocated rather than passed down). This approach ensures comprehensive tracking of sensitive data flow throughout the program's execution path.

Referring to Listing 4, our intra-procedural analysis begins with `bar`, which uses a system call `open` at line 24. Initially, we add `flag` and `path` to the sensitive argument set of `open` and create individual sets for each variable in `bar` (`flag`, `flag2`, and `path`). Since `flag`'s value depends on `flag2`, we add `flag` to `flag2`'s set. Upon completing the analysis of `bar`, `flag2`'s set includes `flag`, leading us to conclude that `flag`, `flag2`, and `path` are all sensitive data. The analysis then proceeds to `foo`. `flag` is added to the sensitive set, as it was identified as sensitive in `bar`. The analysis concludes here since `flag` is declared within `foo` and its set contains only itself. Scynteg instruments `api_record` to record the value as a shadow copy before updating data in the sensitive argument set. For the data passed across function calls, Scynteg adds a `api_record_reg` before the caller invokes the callee. Scynteg adds a `api_check_reg` upon entering the callee function. Scynteg instruments `api_check` to the program before a system call is made.

4.2.5 Shadow Memory Integrity. We leverage a re-randomization technique proposed by previous work [54] to protect shadow memory. We periodically shuffle the placement of shadow memory in the address space, thereby reducing the likelihood of successful corruption by attackers. Before each invocation of sensitive system calls, we shuffle the location of the shadow memory. We call the function `mmap` to create a new random memory location to shuffle the shadow memory, followed by invoking `mremap` to move the old memory to the new one. The `mremap` function remaps the virtual memory address, rendering the old memory inaccessible.

5 Scynteg Implementation

We implemented a Scynteg 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. Given Arm's increasing popularity, our implementation supports Arm-based platforms. 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 Arm64 backend. In addition, we

added two LLVM passes, respectively, for forward-control-flow protection and argument integrity protection. For the former, we leverage Arm's PA to simplify implementation efforts. The changes for LLVM in total require 2,790 LoC. Finally, we added a runtime library to support the extended LLVM with 443 LoC.

5.1 Loadable Kernel Module

The monitor module reads the section table [32] 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` and `__clone` functions. Scynteg's monitor supports unwinding and callsite checking against statically and dynamically linked program binaries. For programs that use dynamic linking, the monitor checks whether the program to be executed contains a `.dynamic` section at program load time. If yes, it traverses the `.dynamic` section to identify all shared libraries that the program dynamically links with. The monitor searches for shared library objects from pre-defined paths and collects their `.eh_frame` sections. A program can also make system calls such as `dlopen` to load library objects at runtime. The libraries these system calls load do 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 callsite checking, the monitor could identify if the call target is within the program's `.plt` section (i.e., the callee is in an external library). If so, the monitor refers to the `.plt` section and resolve them to get the actual address of the callee.

5.2 Extending LLVM for Forward-edge CFI Protection

We utilize the pointer authentication (PA) [33, 47] feature provided by recent Arm processors to secure forward-edge control flow integrity. We could also leverage Intel's Control-flow Enforcement (CET) [17] technology to protect the x86-based implementation. First, we aim to secure the integrity of function pointers using PA. Additionally, we use PA to tag pointers to C++'s virtual method table (Vtable) to protect the integrity of virtual method invocations in C++ programs. We introduced a new LLVM pass to instrument PA instructions to the program. This LLVM pass instruments PA's signing instructions before storing an address value to (1) a value to 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.

Selecting PA modifier. Previous pointer authentication (PA) approaches have used either a constant modifier [6] for all signed pointers or assigned unique 64-bit modifiers [24] for each protected pointer. The former is vulnerable to pointer reuse attacks, while the latter incurs significant overhead in storing and managing modifiers and metadata. In a pointer reuse attack, an attacker exploits a leaked PA-signed pointer by reusing it at any authentication site with the same PA modifier used at signing. This is particularly problematic in fixed modifier schemes. Scynteg aims to prevent system call function hijacking while minimizing storage and management overhead. For protecting function pointers, Scynteg uses a function's signature as its PA modifier, enhancing efficiency and security. This method is robust against pointer reuse attacks, based on our observation that attackers typically attempt to replace a program's custom function pointer with a system call function address. Since custom and system call function signatures usually differ, using the function's signature as the PA modifier proves effective. On the other hand, Scynteg utilizes the address of the VPointer as a modifier for signing the VPointer, binding the pointer's location and value. Because VPointers do not change much over the program's lifetime, binding pointers to addresses ensures that the pointer value remains unmodified, preventing that location from being corrupted.

We extract function signatures for signing function pointers via the `FunctionType` class. 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 access the VTable pointers in the LLVM front-end. In these functions, we use the address of the VTable pointer as a PA modifier, as mentioned in Section 4.1.1. We introduced new PA-specific LLVM intrinsics. These intrinsics pass PA modifiers to the backend. The information is transferred to the emitted LLVM MIR and used when generating Arm PA instructions.

Pointer Signing/Authentication. 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. We then insert an intrinsic call to `pacia` with this information as arguments.

On the other hand, to ensure portability, our approach refrains from recompiling libraries such as libc.so. Consequently, if a signed pointer is carried as an argument by functions like `sigaction` defined in a library 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 LLVM's `TargetLibraryInfo` class to discern between user-defined and library function calls. The instrumentation collects essential information, such as the pointer value and the function signature, and inserts an intrinsic call to `autia` with this information as an argument.

We also 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, 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 use a function pointer, such as an indirect call instruction. We collect the same information as when signing a pointer. 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) and `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 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.

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

6 Performance Evaluation

We ran all benchmarks on Apple's M1 processor [5], which supports the ARMv8.3 architecture with ARM PA instructions. Specifically, we used an Apple Mac Mini M1 [4] equipped with 8GB of DRAM, four big and four small cores. To test our loadable kernel module, we replaced macOS with Asahi Linux. We did not enable compiler optimizations for all benchmarks to ensure a fair comparison.

We evaluated a scenario where users employ robust security measures, such as a shadow stack, to protect applications. While Arm has introduced a hardware-based shadow stack called Guarded Control Stack (GCS) [34] in the Arm v9-A architecture, it is not yet supported in any existing Arm processors. We extended the implementation of LLVM's Shadow Call Stack (SCS) [50] to simulate GCS's performance impact. This extension replicates the runtime operations of GCS, providing comparable stack protection, and conservatively evaluates the performance implications of GCS in current hardware environments.

We evaluated network applications from Table 3. These applications were chosen for their prevalence and susceptibility to attacks. They perform intensive I/O operations via system calls, making them suitable candidates for our study. All applications run on the Arm server to communicate with a remote client on a separate x86-based machine in the same local network. For these applications, we tested 20 runs and presented the average results. Furthermore, we evaluated the performance of C++ programs using the SPECCPU2017 [10] benchmark suite to evaluate the PA-based protection of VPointer. We tested five benchmarks written in C++ and one written in the mix of C and C++.

Table 4 presents the benchmarking results with different combinations of Scynteg features. When the Unwinder (UW) and Argument Integrity (AI) are employed, the performance overhead for Lighttpd is approximately 0.49% and 0.61%, respectively. Scynteg incurs a negligible overhead of 0.78% when all protections are applied. As shown in Table 4, the resulting overhead for NGINX when utilizing all the full Scynteg features is modest; the overhead never exceeded a 1.72% degradation compared to the baseline unprotected NGINX. On the other hand, SQLite incurred much higher overhead with Scynteg's protection feature on UW. Compared to Lighttpd and NGINX, the SQL server executes more system calls write/read.

We present the results of SPECCPU2017 in Table 5. When all protection features are enabled, the resulting overhead is 7.38% higher than the unprotected baseline. VPointer protection incurred higher overhead than UW in SPECCPU.

Table 3: Network Application Benchmarks.

Benchmark	Description
Lighttpd [29]	Used wrk [52] to report Requests per second (Req/sec) in a 20-second duration to make HTTP requests for a 6,227-byte static webpage with one worker thread.
NGINX [2]	Used the same wrk setup as Lighttpd to measure Req/sec, except with 8 worker threads.
SQLite [36]	Used sqlite-bench [1] to report the performance of read and write operations. For read operations, we tested `readrandom` to make N reads in random order. For write operations, we tested `fillrandom` to write N values in random key order in async mode. We ran these benchmarks with the default settings and set N to 1,000,000. We reported the number of microseconds per operation (micros/op).

Table 4: Network Application Performance

UW: Unwinding AI: Argument Integrity for System Call Argument
PA: Pointer Authentication.

Application	Unprotected	UW	UW+AI	UW+AI+PA
Lighttpd	14470.19	14399.38	14382.58	14356.82
NGINX	13451.34	13406.41	13221.11	13220.33
SQLite write	13.48	51.76	51.95	53.21
SQLite read	3.78	8.95	9.00	9.04

7 Security Analysis

We analyzed the efficacy of Scynteg's protection against attacks in real-world, including ROP, SROP [9], CHOP [19], CVEs[38–40, 42, 43], and advanced attack techniques proposed in previous work [20, 48, 51]. Scynteg significantly raises the bar for attackers to succeed in making sensitive system calls.

ROP, CHOP, and SROP. To carry out an ROP attack to invoke system calls, the attacker aims to place gadgets on the stack and overwrite return addresses. Scynteg's monitor detects such stack corrupt via backward CFI protection. Moreover, ROP entails executing gadgets to set the desired system call arguments. Scynteg prevents the compromise as the overwritten argument value mismatches with the ones stored in the shadow memory.

Table 5: Benchmark numbers for SPEC CPU2017 (measures the number of seconds per task).

UW: Unwinding AI: Argument Integrity for System Call Argument PA: Pointer Authentication

Protection	520.ommnetpp_r	523.xalancbmk_r	531.deepsjeng_r	541.leela_r	508.namd_r	511.povray_r
Unprotected	1503	1455	758	2210	1056	1377
UW	1518 (0.99%)	1467 (0.82%)	770 (1.58%)	2215 (0.23%)	1061 (0.47%)	1404 (1.96%)
UW+AI	1553 (3.33%)	1484 (1.99%)	784 (3.56%)	2323 (5.11%)	1065 (0.85%)	1430 (3.85%)
UW+AI+PA	1589 (5.72%)	1537 (5.64%)	802 (5.80%)	2373 (7.38%)	1077 (1.99%)	1478 (7.33%)
PA	1532 (1.93%)	1457 (0.14%)	774 (2.11%)	2206 (0.78%)	1064 (0.76%)	1369 (1.11%)
UW+PA	1539 (2.40%)	1508 (3.64%)	790 (4.22%)	2252 (2.88%)	1074 (1.70%)	1390 (2.66%)

A CHOP attack aims to overwrite the return address on the stack of the vulnerable exception-throwing function, allowing the intended exception handler to hijack control flow. Like ROP, Scynteg could detect such stack corruptions by backward CFI protection. Also, since CHOP involves hijacking a function pointer to redirect the execution flow, Scynteg can detect such compromise by PA. An attacker could exploit SROP that installs a fabricated signal frame in the stack and pops the states from the frame (e.g., PC) to the hardware to hijack control flow and make sensitive system calls. Since SROP involves stack corruption, Scynteg could detect an attacker's attempt via callsite checking.

VPointer Hijacking. We evaluated Scynteg against four VPointer hijacking exploits, and one Counterfeit Object Oriented Programming (COOP) [49] exploit from the CFIXX C++ test suite [37]. The VPointer hijacking exploits overwrite an existing VPointer. COOP crafts class objects allocated from the heap that use malicious VTables to replace benign objects. Both attacks fail the PA authentication imposed by Scynteg.

Other attacks. Function pointer hijacking vulnerabilities from NGINX [20, 48, 51] and Apache [48] enables redirection of program execution to execute sensitive system calls, such as `execve`. Scynteg leverages Arm PA and enforcing argument integrity to protect against function pointer hijacking. Various CVEs [38, 41–43] cause a buffer overflow, while others [39, 40] exploit format string vulnerability that enables arbitrary memory writes. These CVEs allow an attacker to tamper with memory contents. Scynteg identifies compromises against return addresses, code pointers, arguments, and their dependent variables, thwarting any attacker attempting to exploit these vulnerabilities.

Non-Control Data Attacks on System Call Argument. In contrast to control data attacks, which aim to hijack control data such as return addresses or function pointers, non-control data attacks [14] focus on tampering with user identity and configuration data. The authors [14] presented scenarios that compromised user identity and configuration data, resulting in tempering system call argument. An attacker could exploit the format string vulnerability [13] from WU-FTPD [53] to bypass control-data-based protections and hijack the argument of the `setuid` function to escalate kernel privileges. The authors also proposed leveraging a heap overflow vulnerability to manipulate the argument of `execve` to spawn a root shell. Scynteg secures the arguments used by `setuid` and `execve` by identifying inconsistencies in system call arguments between the shadow and regular memory to thwart the attack.

8 Related Work

Debloating and system call filtering. Debloating [3, 45, 46] reduces 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 [26] filters to block any outside this set. Some [12, 18] automate the generation of these filter sets, while [22] 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, Scynteg ensures the execution flow is not hijacked when a process invokes sensitive system calls.

Runtime System Call Protection. Previous work [11, 27, 28] checks a system call control flow graph (CFG) to ensure the integrity of system call usage. Unlike Scynteg, [11, 28] do not secure system call arguments so that attackers could corrupt them. Further, none of the works supports Arm. The CFG that those works rely on could be unavailable in program binaries. 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 [27] hooks sensitive system calls to protect their usage. It relies on Linux's ptrace to monitor and hook system calls. Upon an application that makes a sensitive system call, Bastion context switches to the monitoring process to validate system call usage. Another context switch is needed to return control to the application. The context switches in Bastion result in significant overhead in network applications that frequently invoke system calls. Bastion causes a 95% drop in NGINX's throughput while Scynteg incurs a modest 2% drop.

To protect against system call attacks, 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 only focuses on the call type of the system call itself and not on the call type of the call chain functions that invoke the system call. For example, in AOCR Nginx Attack 2 [48], the author demonstrates compromising Nginx by overwriting a function pointer to a function that invokes a sensitive system call, which is illegal to invoke through an indirect function call. In such a situation, Bastion would conclude that there is no attack because the call type of the system call is legal, thereby failing to detect the illegal call type of the caller function. In contrast, Scynteg enhances security by protecting function pointer integrity through Pointer Authentication (PA) and argument integrity. These

approaches can detect all attempts to hijack a function pointer, offering a more comprehensive defense against such attacks.

Bastion ensures argument integrity for system calls by copying each argument into a secure region during write operations. Before a system call is invoked, Bastion verifies that the value in the secure region matches the value in regular memory. It uses a prior approach [25] that leverages Intel's Memory Protection Keys (MPK) [16] to write-protect the secure region. MPK is unavailable on Arm. In contrast, Scynteg uses a software-based re-randomization approach to protect the secure shadow memory region, offering greater portability.

Pointer Integrity Protection. Previous work [31, 35] proposed various schemes to protect code pointer integrity. SafeStack [31] isolates pointers but not VPointers, while CCFI [35] uses software-based MACs to protect function and VPointers. CFIXX focuses on preventing object-type corruption to ensure the integrity of virtual function dispatch. Unlike Scynteg, these approaches do not leverage hardware support like Arm PA, resulting in higher performance overhead. Other works [24, 30], such as PARTS [30] and PACTight [24], utilize PA for pointer integrity. However, PARTS does not protect VPointers, and PACTight's complex scheme for managing random tags leads to significant overhead (up to 9% slowdown for C++ SPECCPU benchmarks). In contrast, Scynteg uses function signatures and locations as PA modifiers, avoiding this complexity and achieving lower overhead (less than 5% for SPECCPU benchmarks).

Unwinding based Approaches. Similar to Scynteg, SLICK [21] uses stack unwinding to detect ROP attacks and safeguard backward-edge CFI. However, unlike the callsite checking employed by Scynteg, 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 Scynteg's kernel module as future work.

SROP and CROP. Previous work [9] proposed adding a canary to the signal frame from the stack to identify corruptions. Previous work [19] proposed checking the stack canary before the unwinding process begins for exception handling to mitigate CHOP. These approaches complement Scynteg and could be incorporated to bolster security.

9 Concluding Remarks

This work introduces Scynteg, a new framework to secure applications' system call usage. Scynteg leverages a runtime and static time approach that consists of a secure loadable kernel module and an LLVM-based compiler to collectively enforce the integrity of a program's system calls' control flow and arguments. Scynteg offers modularized application protection. Scynteg's module can be independently employed by production systems to enhance system call protections without recompilation of application programs. If the program source is available, Scynteg's compiler can be used to instrument programs to further bolster their safety. This flexibility allows for tailored security measures for different deployment scenarios to meet custom security, functionality, and performance demands. Our Scynteg prototype strengthens application security for Arm-based Linux environments, leveraging emerging hardware extensions to effectively protect against advanced return-oriented-based attacks in the real world while incurring modest performance overhead to widely used applications.

Acknowledgement

We thank all anonymous reviewers for your insightful comments. This research work was supported by Delta Electronics, Inc. under research grant 111HT907011.

References

[1] 2018. A SQLite3 benchmark tool. https://github.com/ukontainer/sqlite-bench.
[2] 2022. NGINX web server. https://nginx.org.
[3] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3359789.3359823
[4] Apple. 2020. Apple Mac Mini M1. https://www.apple.com/shop/buy-mac/mac-mini/applem1-chip-with-8-core-cpu-and-8-core-gpu-256gb.
[5] Apple. 2020. Apple unleashes M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.
[6] Apple Inc. May 2022. Apple Platform Security. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf.
[7] Arm Developer. 2014. Execute Never. https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Memory-attributes/Execute-Never?lang=en.
[8] Roberto Avanzi. 2017. The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology* (2017), 4–44.
[9] Erik Bosman and Herbert Bos. 2014. Framing Signals—A Return to Portable Shellcode. *IEEE Symposium on Security and Privacy* (2014).
[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.
[11] Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. 2022. SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems. *arXiv preprint arXiv:2202.13716* (2022).
[12] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*. 139–151.
[13] CERT Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD [n. d.]. https://vuls.cert.org/confluence/display/historical/CERT+Advisory+CA-2001-33+Multiple+Vulnerabilities+in+WU-FTPD.
[14] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD) *(SSYM'05)*. USENIX Association, 12.
[15] DWARF Debugging Information Format Committee. [n. d.]. DWARF Debugging Information Format, Version 4. https://dwarfstd.org/doc/DWARF4.pdf.
[16] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.
[17] Intel Corporation. 2020. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.
[18] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 459–474.
[19] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. 2017. Let Me Unwind That For You: Exceptions to Backward-Edge Protection. *Network and Distributed System Security Symposium* (2017).
[20] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).
[21] Yangchun Fu, Junghwan Rhee, Zhiqiang Lin, Zhichun Li, Hui Zhang, and Guofei Jiang. 2016. 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*. Springer, 71–94.
[22] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. 1749–1766.

[23] H. Hu, S. Shinde, S. Adrian, Z. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 969–986. https://doi.org/10.1109/SP.2016.62

[24] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2022. Tightly Seal Your Sensitive Pointers with {PACTight}. In *31st USENIX Security Symposium (USENIX Security 22)*. 3717–3734.

[25] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1612–1626. https://doi.org/10.1145/3460120.3485376

[26] Jake Edge. 2012. A library for seccomp filters. https://lwn.net/Articles/494252/.

[27] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 2023. 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*. 528–541.

[28] Lap Chung Lam and Tzi-cker Chiueh. 2004. 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*. Springer, 1–20.

[29] lighttpd. 2024. Lighttpd web server. https://www.lighttpd.net/.

[30] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.

[31] Linux Foundation. [n. d.]. Special Sections. https://refspecs.linuxfoundation.org/LSB_1.2.0/gLSB/specialsections.html.

[32] Linux Foundation. 2001. Section Header. https://refspecs.linuxbase.org/elf/gabi4+/ch4.sheader.html.

[33] Mark Rutland. September 14, 2017. ARMv8.3 Pointer Authentication. https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf.

[34] Martin Weidmann. September 29, 2022. ARM v9-A. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022.

[35] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[36] SQLite Consortium members. [n. d.]. SQLite. https://www.sqlite.org/index.html.

[37] Nathan Burow. 2018. CFIXX C++ test suite. https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite.

[38] National Institute of Standards and Technology. National Vulnerability Database. 2015. 2013. https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[39] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2012-0809 2012. https://nvd.nist.gov/vuln/detail/CVE-2012-0809.

[40] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2015-8617 2015. https://nvd.nist.gov/vuln/detail/CVE-2015-8617.

[41] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2019-3822 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-3822.

[42] National Institute of Standards and Technology. National Vulnerability Database. 2016. CVE-2016-10190 [n. d.]. https://nvd.nist.gov/vuln/detail/CVE-2016-10190.

[43] National Institute of Standards and Technology. National Vulnerability Database. 2016. CVE-2016-10191 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-10191.

[44] PaX. 2003. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.

[45] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[46] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[47] Qualcomm Technologies, Inc. January 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[48] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. *Network and Distributed System Security Symposium* (2017).

[49] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[50] The Clang Team. 2024. ShadowCallStack. https://releases.llvm.org/16.0.0/tools/clang/docs/ShadowCallStack.html.

[51] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. *ACM SIGSAC Conference on Computer and Communications Security* (2017).

[52] Will Glozer. 2019. a HTTP benchmarking tool. https://github.com/wg/wrk.

[53] WU-FTPD Development Group 2003. https://web.archive.org/web/20080324003630/http://www.wu-ftpd.org/.

[54] Changwei Zou, Yaoqing Gao, and Jingling Xue. 2022. Practical Software-Based Shadow Stacks on x86-64. *ACM Trans. Archit. Code Optim.* 19, 4 (oct 2022). https://doi.org/10.1145/3556977