PACStack: an Authenticated Call Stack

Hans Liljestrand
University of Waterloo, Canada
hans@liljestrand.dev

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

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

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

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

Abstract

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

We present *authenticated call stack* (ACS), an approach that uses chained message authentication codes (MACs). Our prototype, PACStack, uses the ARM general purpose hardware mechanism for pointer authentication (PA) to implement ACS. Via a rigorous security analysis, we show that PACStack achieves security comparable to hardware-assisted shadow stacks *without requiring dedicated hardware*. We demonstrate that PACStack's performance overhead is small ($\approx$3%).

1 Introduction

Traditional code-injection attacks are ineffective in the presence of W$\oplus$X policies that prevent the modification of executable memory [49]. However, code-reuse attacks can alter the run-time behavior of a program without modifying any of its executable code sections. Return-oriented programming (ROP) is a prevalent attack technique that corrupts function return addresses to hijack a program's control flow. ROP can be used to achieve Turing-complete computation by chaining together existing code sequences in the victim program. To prevent ROP, return addresses must be protected when stored in memory. At present, the most powerful protection against ROP is using an *integrity-protected shadow stack* that maintains a secure reference copy of each return address [1]. Integrity of the shadow stack is ensured by making it inaccessible to the adversary either by randomizing its location in memory or by using specialized hardware [29].

Recent software-based shadow stacks show reasonable performance [10], but are vulnerable to an adversary capable of exploiting memory vulnerabilities to infer the location of the shadow stack. To date, only hardware-assisted schemes, such as Intel CET [29], achieve negligible overhead without trading off security. But employing such a custom hardware mechanism incurs development and deployment costs.

Recent ARM processors include support for pointer authentication (PA); a hardware extension that uses tweakable message authentication codes (MACs) to sign and verify pointers [4]. One initial use case of PA is the authentication of return addresses [45]. However, current PA schemes are vulnerable to *reuse attacks*, where the adversary can reuse previously observed valid protected pointers [35]. Prior work [35, 45] and current implementations by GCC [1] and LLVM [2] mitigate reuse attacks, but cannot completely prevent them.

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

- ACS, a new approach for **precise verification of function return addresses** by chaining MACs (Section 4).
- PACStack, an LLVM-based realization of ACS using ARM PA **without requiring additional hardware** (Section 5).
- A systematic evaluation of PACStack security, showing that its **security is comparable to shadow stacks** (Section 6).
- Demonstrating that the **performance overhead** of PACStack **is small** ($\approx$3%) (Section 7).

PACStack and associated evaluation code is available as open source at `https://pacstack.github.io`.

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

2 Background

2.1 ROP on ARM

In ROP, the adversary exploits a memory vulnerability to manipulate return addresses stored on the stack, thereby altering the program's backward-edge control flow. ROP allows Turing-complete attacks by chaining together multiple gadgets, i.e., adversary-chosen sequences of pre-existing program instructions that together perform the desired operations. ARM architectures use the link register (`LR`) to hold the current function's return address. `LR` is automatically set by the *branch with link* (`bl`) or *branch with link to register* (`blr`) instructions that are used to implement regular and indirect function calls. Because `LR` is overwritten on call, non-leaf functions must store the return address onto the stack. This opens up the possibility of ROP on ARM [30].

2.2 ARM Pointer Authentication

The ARMv8.3-A PA extension supports calculating and verifying pointer authentication codes (PACs) [4]. PA is at present deployed in the Apple A12, A13, S4, and S5 systems-on-chip (SoCs) and is going to be available in all upcoming ARMv8.3-A and later SoCs. A `pac` instruction calculates a keyed tweakable MAC, $H_K(A_P, M)$, over the address A_P of a pointer P using a 64-bit modifier M as the tweak. The resulting authentication token, referred to as a PAC, is embedded into the unused high-order bits of P. It can be verified using an `aut` instruction that recalculates $H_K(A_P, M)$, and compares the result to P's PAC.

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

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

[3] https://kernelnewbies.org/Linux_5.0#ARM_pointer_authentication

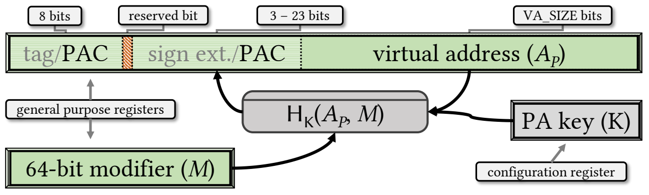


Figure 1: PA uses a pointer authentication code (PAC) based on the pointer's address, a modifier, and a key.

```
prologue:
  paciasp               ; sign LR using SP          ❶
  str LR, [SP]          ; push LR onto stack        ❷
  ...
epilogue:
  ldr LR, [SP]          ; pop stack onto LR         ❸
  retaa                 ; verify LR and return      ❹
```

Listing 1: The `-mbranch-protection` feature in GCC and LLVM/Clang uses PA to sign (❶) and verify (❹) the return address in `LR`. PA does not access memory directly, the `LR` value is stored (❷) and loaded (❸) conventionally.

2.2.1 PA-based return address protection

PA-based return address protection is implemented as part of the `-mbranch-protection` feature of GCC and LLVM/-Clang. [4] An authenticated return address is computed with `paciasp` (❶ in Listing 1) and verified with `retaa` (❹). These instructions use the instruction key *A* and the value of stack pointer (`SP`) as the modifier. The PA-keys are protected by hardware; consequently an adversary has to resort to guessing the correct PAC for a modified return address.

The `-mbranch-protection` feature and other prior PA-based solutions are vulnerable to *reuse attacks* where an adversary replaces a valid authenticated return address with another authenticated return address previously read from the process' memory. For a reused PAC to pass verification, both the original and replacement PAC must have been computed using the same PA key and modifier. This applies to any PA scheme, not only authenticated return addresses. Using the `SP` value as a modifier reduces the set of interchangeable pointers, but still allows reuse attacks when `SP` values coincide. Reuse attacks can be mitigated, but not completely prevented, by further narrowing the scope of modifier values [35].

3 Adversary model and requirements

In this work, we consider a powerful adversary, $\mathcal{A}$, with arbitrary control of process memory but restricted by a W$\oplus$X policy that prevents modification of code pages. This adversary model is consistent with prior work on run-time attacks [49]. We limit $\mathcal{A}$ to user space; thus $\mathcal{A}$ cannot read or modify kernel-managed registers such as the PA keys.

[4] https://gcc.gnu.org/gcc-9/changes.html and https://reviews.llvm.org/D51429

We make the following assumptions about the system:

A1 *A W⊕X policy* protects code memory pages from modification by non-privileged processes. All major processor architectures, including ARMv8-A, support W⊕X.

A2 *Coarse-grained forward-edge control-flow integrity (CFI)* that restricts forward control-flow transfers to a set of valid targets. Specifically, we assume that indirect function-calls always target the beginning of a function and that indirect jumps to arbitrary addresses is infeasible. This property is satisfied by several preexisting software-only CFI solutions with reasonable overhead [1, 18, 31, 37], as well as with negligible overhead by using hardware-assisted mechanisms like ARM PA [35], branch target indicators [4], or TrustZone-M [5, 39]. In particular, a minimal PA scheme using a constant (e.g., `0x0`) modifier fulfills this assumption.

This adversary model allows $\mathcal{A}$ to modify any pointer in data memory pages. In particular, $\mathcal{A}$ can modify function return addresses while they reside on the program call stack. **A2** and **A1** prevent $\mathcal{A}$ from tampering with ACS instrumentation (Section 6.3). Our goal is to thwart $\mathcal{A}$ who modifies function return addresses in order to hijack the program control flow. We define the following requirements:

R1 *Return address integrity*: Detect if a function return address has been modified while in memory.

R2 *Memory disclosure tolerance*: Remain effective even when $\mathcal{A}$ can read the entire process address space.

R3 *Compatibility*: Be applicable to typical (standard-compliant) C code without source code modifications.

R4 *Performance*: Impose only minimal run-time performance and memory overhead, while meeting **R1**–**R3**.

As in prior work on CFI, we do not consider non-control data attacks [12], such as data-oriented programming (DOP) [27].

4 Design: authenticated call stack

In this section we present our general design for an authenticated call stack (ACS). In Section 5, we present our implementation that efficiently realizes ACS using ARM PA. Our key idea is to provide a modifier for the return address by cryptographically binding it to *all previous return addresses in the call stack*. This makes the modifier statistically unique to a particular control-flow path, thus preventing reuse-type attacks and allowing *precise verification of return addresses*. The return addresses $ret_i, i \in [0,n]$ (where n is the depth of the call stack in terms of active function records) must be stored on the stack, where $\mathcal{A}$ can modify them by exploiting memory vulnerabilities. ACS protects these values by computing a series of *chained* authentication tokens $auth_i, i \in [0,n]$ that cryptographically bind the last $auth_n$ to all return addresses $ret_i, i \in [0, n-1]$ stored on the stack (Figure 2). Only the MAC key and the last authentication token $auth_n$ must be stored securely to ensure that previous $auth$ tokens and return addresses can be correctly verified when unwinding the call stack (**R1**). We use a tweakable MAC function H_K to generate a b-bit authentication token $auth_i$:

$$auth_i = \begin{cases} \mathtt{H}_\mathtt{K}(ret_i, auth_{i-1}) & \text{if } i > 0 \\ \mathtt{H}_\mathtt{K}(ret_i, 0) & \text{if } i = 0 \end{cases}$$

$auth_n$ is maintained in a register unmodifiable by $\mathcal{A}$. Figure 3 shows how authentication tokens and return addresses are stored on the call stack. On function calls, $auth_i$ is retained across the call to the callee, which calculates $auth_{i+1}$ and stores both $auth_i$ and the corresponding return address ret_{i+1} on its stack frame. On return, $auth'_{i-1}$ and ret'_i values are loaded from the stack and are verified by comparing $\mathtt{H}_\mathtt{K}(auth'_{i-1}, ret'_i)$ to $auth_i$. If the results differ, then one or both of the loaded values have been corrupted (**R1**). Otherwise, they are valid—i.e., $auth'_{i-1} = auth_{i-1}$ and $ret'_i = ret_i$—in which case $auth_i$ is replaced with the verified $auth_{i-1}$ in the secure register before the function returns to ret_i.

For compactness, we can combine $auth_i$ and ret_i, into an *authenticated return address*, $aret_i$:

$$aret_i = auth_i \parallel ret_i, \text{where}$$
$$auth_i = \begin{cases} \mathtt{H}_\mathtt{K}(ret_i, aret_{i-1}) & \text{if } i > 0 \\ \mathtt{H}_\mathtt{K}(ret_i, 0) & \text{if } i = 0 \end{cases}$$

We call $auth_i$ and the corresponding $aret_i$ *valid* if $auth_i = \mathtt{H}_\mathtt{K}(ret_i, aret_{i-1})$ for some given $aret_{i-1}$.

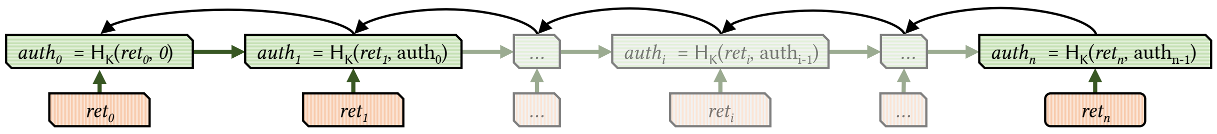
4.1 Securing the authentication token

The current authenticated return address $aret_n$, is secured by keeping it exclusively in a CPU register which we call the *chain register* (`CR`). Note that reserving exclusive use of a register is also a requirement for current shadow stack implementation for the 64-bit ARM architecture [14] and has been proposed for shadow stacks on the x86 architecture [10].

ACS protects the integrity of backward-edge control-flow transfers. Combined with coarse-grained forward-edge CFI (Assumption **A2**), it ensures that: 1) immediately after function return, the $aret_n$ in `CR` is valid, 2) at function entry the $aret_{n-1}$ stored in `CR` is valid, and 3) `CR` is always used as or set to a valid $aret$. This ensures that token updates are done securely, and that the ACS instrumentation cannot be bypassed or used to generate arbitrary authenticated return addresses.

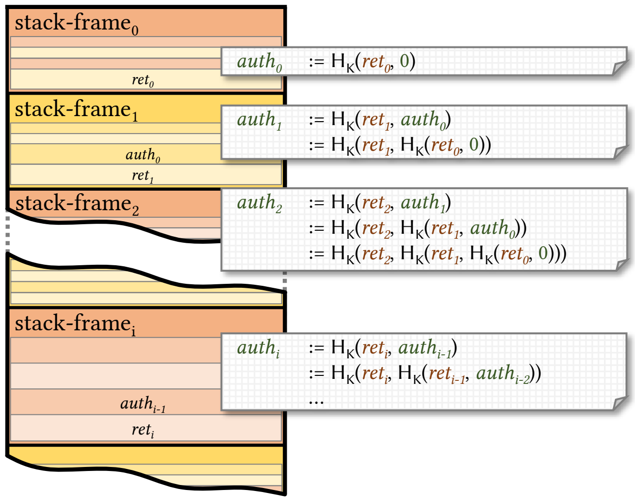
4.2 Mitigating hash-collisions

Though $aret_n$ is protected by hardware, the size b of the authentication token $auth$ can be limited by the implementation. Using a PAC as the token would typically limit it to 16 bits. This is significant, as collisions can be found after $\mathcal{A}$ has



auth_0 = H_K(ret_0, 0)
auth_1 = H_K(ret_1, auth_0)
...
auth_i = H_K(ret_i, auth_i-1)
...
auth_n = H_K(ret_n, auth_n-1)
ret_0
ret_1
...
ret_i
...
ret_n

Figure 2: ACS is an chained MAC of tokens $auth_i, i \in [0, n-1]$ that are cryptographically bound to the corresponding return addresses, $ret_i, i \in [0, n]$, and the last $auth_n$.



stack-frame_0
ret_0
auth_0 := H_K(ret_0, 0)
stack-frame_1
auth_0
ret_1
auth_1 := H_K(ret_1, auth_0)
:= H_K(ret_1, H_K(ret_0, 0))
stack-frame_2
auth_2 := H_K(ret_2, auth_1)
:= H_K(ret_2, H_K(ret_1, auth_0))
:= H_K(ret_2, H_K(ret_1, H_K(ret_0, 0)))
stack-frame_i
auth_i-1
ret_i
auth_i := H_K(ret_i, auth_i-1)
:= H_K(ret_i, H_K(ret_i-1, auth_i-2))
...

Figure 3: ACS stores return addresses and intermediate authentication tokens, $auth_i, i \in [0, n-1]$, on the stack. Only the last token ($auth_n$) needs to be securely stored.

seen, on average, approximately $1.253 \cdot 2^{b/2}$ tokens [47, Section 1.4.2] (e.g., 321 tokens for $b = 16$). Despite this, we can still prevent $\mathcal{A}$ from *recognizing* collisions (**R2**), thus forcing $\mathcal{A}$ to guess—with a success probability 2^{-b}—which authenticated return addresses yield a collision. The *auth* of any *aret* stored on the stack is masked using a pseudo-random value derived from the previous *aret* value:

$$auth_i = \mathtt{H_K}(ret_i, aret_{i-1}) \oplus \mathtt{H_K}(0, aret_{i-1}).$$

The mask is exclusive-OR-ed with $\mathtt{H_K}(ret_i, aret_{i-1})$ after it is generated and before it is authenticated, thereby preventing $\mathcal{A}$ from identifying opportunities for pointer reuse. We discuss the security of masking in Section 6.2.1.

4.3 Mitigating brute-force guessing

A brute force attack where $\mathcal{A}$ guesses an *auth* token succeeds with probability p for a b-bit *auth* after $\frac{\log(1-p)}{\log(1-2^{-b})}$ guesses, provided that a failed guess terminates the program and subsequent program runs use a new key to generate *auth* tokens. This assumption is similar to prior PA-based solutions [35] and is consistent with current PA behavior in Linux 5.0. However, if pre-forked or multithreaded programs share the key, $\mathcal{A}$ can target a vulnerability in a sibling. Unless a failed authentication terminates the entire process tree, $\mathcal{A}$ can then attempt a new guess against another sibling process without resetting the key. In this scenario, 2^{b-1} guesses on average are enough to obtain a modifier with respect to which some combination of pointer and authentication token is valid. Since this modifier becomes the next authenticated return address, the process can be repeated to use the injected address. Because the two guesses can be done separately using a divide-and-conquer strategy, this requires on average 2^b guesses to allow $\mathcal{A}$ to jump to an arbitrary address, rather than 2^{2b} that are needed when the guesses are independent.

Liljestrand et al. [35] recommend hardening pre-forking and multi-threaded applications against guessing attacks by having the application restart all of its processes if the number of PAC failures in child processes exceeds a pre-defined threshold. We recommend an alternative mitigation specific to ACS: *"re-seeding"* the *auth* calculation after a fork or thread creation. For example, calculating $auth_0 = \mathtt{H_K}(ret_0, init)$ where *init* corresponds to the process or thread ID. This solution is straightforward to apply to threads, as a return from the function starting the thread causes the thread to exit. Crucially, re-seeding prevents a divide-and-conquer guessing strategy and requires on average 2^{2b} guesses. Therefore, the ACS for the thread stacks can be made disjoint from the main ACS chain. However, forked processes may use *auth* tokens in stack frames inherited from the parent process. If a child process never returns to inherited stack frames, re-seeding any new *auth* tokens beyond the point of the fork is sufficient. However, if the child process returns to inherited stack frames, the ACS must be re-seeded starting from $auth_0$ by rewriting any *auth* tokens in pre-existing stack frames; similar to some stack canary re-randomization schemes [25, 43].

4.4 Irregular stack unwinding

The C standard includes the `setjmp` / `longjmp` programming interface, which can be used to add exception-like functionality to C. The `longjmp` C function executes a non-local jump to a prior calling environment stored using the `setjmp` function. At `setjmp`, callee-saved registers (whose values are guaranteed to persist through function invocations), as well as the stack pointer `SP`, and the return address are stored in the given `jmp_buf` buffer. Calling `longjmp` using an expired buffer, i.e., after the corresponding `setjmp` caller has returned, results in undefined behavior (the implications of this are discussed in Section 9.1). Because `jmp_buf` also stores the last authenticated token, ACS needs a mechanism to ensure its integrity when using `setjmp` and `longjmp`.

While in memory, the integrity of jmp_buf cannot be guaranteed. Nonetheless, the stored $auth_i$ is bound to the corresponding $auth_{i-1}$ on the setjmp caller's stack. This ensures that longjmp always restores a valid ACS state. To limit the set of values $\mathcal{A}$ can inject into jmp_buf, we replace the setjmp return address ret_b in jmp_buf with $aret_b$, defined as:

$$aret_b = (\mathtt{H_K}(ret_b, auth_i) \parallel ret_b) \oplus \mathtt{H_K}(\mathtt{SP}_b, auth_i),$$

where $\mathtt{SP}_b$ is the SP value stored in jmp_buf. When executing longjmp, $aret_b$ is recalculated based on the buffer values to verify that the stored $auth_i$ was stored by a setjmp. $\mathcal{A}$ cannot generate the $aret_b$ value for an arbitrary $auth_i$, nor replace $aret_b$ with a previously observed $auth_i$. But, since longjmp explicitly allows jumping to prior states, ACS cannot ensure that the target is the *intended one*, i.e., $\mathcal{A}$ could substitute the correct jmp_buf with another. Shadow stacks share a similar limitation [17], and cannot guarantee that the intended state has been reached, only that the return address (and stack pointer) in that state is intact.

5 Implementation: PACStack

We present PACStack, an ACS realization using ARMv8.3-A PA. PACStack is based on LLVM 9.0 and integrated into the 64-bit ARM backend. PACStack modifies the AArch64FrameLowering such that the function stores and loads $aret_{n-1}$ during FrameSetup and FrameDestroy, respectively. We also modify the AArch64RegisterInfo to ensure that the register holding $aret_n$, chain register (CR), is reserved for PACStack use. Our current implementation uses a Intermediate Representation (IR) pass to mark all functions for instrumentation, whereas the backend then performs instrumentation based on the function attribute.

The current authenticated return address is securely stored in CR. Because the unprotected return address ret_i is never stored on the stack, $\mathcal{A}$ is limited to manipulating the earlier authenticated return addresses on stack, i.e., $aret_i, i \in [0, n-1]$. An authenticated return address must therefore pass two authentications before use: first when being restored from the stack, and second, when being used as the target of a function return. We discuss the security implications in Section 6.

PACStack uses the pacia and autia instructions to efficiently calculate and verify authenticated return addresses (Listing 2, ③ and ⑥). The result of pacia is $aret_i$ which is stored in the link register (LR, ③) and moved to CR (④):

$$\mathtt{LR} \leftarrow aret_i = \begin{cases} \mathtt{pacia}(\mathtt{LR} = ret_i, \mathtt{CR} = aret_{i-1}) & \text{if } i > 0 \\ \mathtt{pacia}(\mathtt{LR} = ret_i, \mathtt{CR} = init) & \text{if } i = 0 \end{cases}$$

The corresponding verification (⑤ and ⑥) are defined as:

$$\mathtt{LR} \leftarrow \mathtt{autia}(\mathtt{LR} = aret_i, \mathtt{CR}) = \begin{cases} ret_i & \text{if } \mathtt{H_K}(ret_i, \mathtt{CR}) = auth_i \\ ret_i^* & otherwise, \end{cases}$$

```
prologue:
  str   X28, [SP, #-32]!   ; stack ← aret_{i-1}         ①
  stp   FP,  LR, [SP,#16]  ; stack ← frame-record       ②
  pacia LR,  X28           ; LR ← aret_i                ③
  mov   X28, LR            ; CR ← aret_i                ④
  ...
epilogue:
  mov   LR,  X28          ; LR ← aret_i
  ldr   FP,  [SP, #16]    ; skip ret'_i in frame-record
  ldr   X28  [SP], #32    ; CR ← aret'_{i-1} from stack  ⑤
  autia LR,  X28          ; LR ← (ret_i or ret*_i)       ⑥
```

Listing 2: At function entry, PACStack stores $aret_{i-1}$ on the stack (①) and generates a new $aret_i$ (③) which is retained in CR (④). Before return, $aret_{i-1}$ is loaded from the stack (⑤) and verified against $aret_i$ (⑥). Verification failure sets LR to an invalid address ret_i^* and causes a fault on return.

where autia will automatically handle verification errors by setting LR to an unusable address ret_i^*. No additional checking is needed; executing a return to ret_i^* causes a address translation fault (Section 2.2). To maintain compatibility (**R3**), PACStack does not modify the frame record (②) and instead stores $aret_{i-1}$ in a separate stack slot (①). This allows, for instance, debuggers to backtrace the call-stack without knowledge of PACStack. PACStack never loads ret_i from the frame record; it always uses $aret_i$ which is securely stored in CR.

5.1 Securing the authentication token

PACStack uses the ARM general purpose register X28 as CR for storing the last authentication token. X28 is a callee-saved register, and so, any function that uses it must also restore the old value before return. By using X28 as CR, PACStack libraries or code can be transparently mixed with uninstrumented code (**R3**). We discuss the security implications of mixing instrumented and uninstrumented code in Section 9.2.

5.2 Mitigating hash collisions: PAC masking

To prevent $\mathcal{A}$ from identifying PAC collisions that can be reused to violate the integrity of the call stack, PACStack masks all authentication tokens values before storing them on the stack. A pseudo-random value is obtained by generating a PAC for address 0x0, $\mathtt{pacia}(0, aret_{i-1})$ (Listing 3 ❶, ❺). By using pacia we efficiently obtain a pseudo-random value that can be directly applied to the authentication token part of *aret* using only an exclusive-or instruction (eor ❷, ❻).

Because this construction uses the same key to generate both authentication tokens and masks, $\mathcal{A}$ must not obtain an $aret_i$ for a $ret_i = \mathtt{0x0}$ and any existing $aret_{i-1}$. PACStack will never generate such *aret* values, as the return address never points to memory address zero. To prevent leaking the mask directly, it is cleared after use (❸, ❼). Consequently no $\mathtt{H_K}(0, x)$ value is visible to $\mathcal{A}$ nor is it possible to pre-compute without the confidential PA key.

```
prologue:
  str    X28, [SP, #-32]! ; stack ← $aret_{i-1}$
  stp    FP,  LR, [SP,#16] ; stack ← frame-record
  mov    X15, XZR          ; X15 ← 0
  pacia  LR,  X28          ; LR ← $aret_i^{unmasked}$
  pacia  X15, X28          ; X15 ← $mask_i$                ❶
  eor    LR,  LR, X15      ; LR ← $mask_i \oplus aret_i^{unmasked}$ ❷
  mov    X15, XZR          ; X15 ← 0                       ❸
  mov    X28, LR           ; CR ← $aret_i$
  ...
epilogue:
  mov    LR,  X28          ; LR ← $aret_i$
  ldr    FP,  [SP, #16]    ; skip $ret_i'$ in frame-record
  ldr    X28, [SP], #32    ; CR ← $aret_{i-1}'$ from stack  ❹
  mov    X15, XZR          ; X15 ← 0
  pacia  X15, X28          ; X15 ← $mask_i$                ❺
  eor    LR,  LR, X15      ; LR ← $mask_i \oplus aret_i$    ❻
  mov    X15, XZR          ; X15 ← 0                       ❼
  autia  LR,  X28          ; LR ← ($ret_i$ or $ret_i^*$)
  ret
```

Listing 3: PACStack masks authentication tokens to hide collisions. The mask is created with $\mathtt{pacia}(0, aret_{i-1})$ (❶), and exclusive-OR-ed with the unmasked $auth_i$ (❷). On return, the masked $auth_i$ is loaded from the stack (❹). The mask is then recreated (❺) and removed from $auth_i$ (❻) before verification. X15 is a scratch register and can be safely used as its value is not retained between function calls.

```
setjmp_wrapper:
  mov    X15, SP;          ; X15 ← $SP_b$
  pacia  X15, X28;         ; X15 ← pacia($SP_b$, $aret_i$)
  pacia  LR,  X28;         ; LR ← pacia($ret_b$, $aret_i$)
  eor    LR,  LR, X15;     ; CR ← $aret_b$
  b      <setjmp>
```

Listing 4: PACStack redirects setjmp calls to our setjmp_wrapper[4] which binds the return address $aret_b$ to $aret_i$ and the SP value before it is stored in jmp_buf.

This approach to masking requires two additional PAC calculations for each function activation. PACStack supports instrumentation with or without masking. We discuss the security of PAC masking in Section 6.2.1.

5.3 Irregular stack unwinding

PACStack binds jmp_buf buffers to the $aret_i$ at the time of setjmp call by replacing the setjmp return address ret_b with its authenticated counterpart $aret_b$ before setjmp stores it to the jmp_buf (Section 4.4). The libc implementation is not modified; instead setjmp / longjmp calls are replaced with the wrapper functions in Listings 4 and 5.

The setjmp_wrapper (Listing 4) replaces the return address in LR with $aret_b$ and then executes setjmp, which stores it in the buffer. The longjmp_wrapper (Listing 5) retrieves $aret_b$, $aret_i$, and the SP values from jmp_buf, verifies their values and writes ret_b into jmp_buf before executing longjmp.

```
longjmp_wrapper:           ; X0 = jmp_buf
  ldr    X28, [X0, #a]     ; CR ← $aret_i'$
  ldr    LR,  [X0, #r]     ; LR ← $aret_b'$
  ldr    X15, [X0, #s]     ; X15 ← $SP_b'$
  pacia  X15, X28;         ; X15 ← pacia($SP_b'$, $aret_i'$)
  eor    X28, X28, X15     ; CR ← $ret_b$
  autia  LR,  X28;         ; LR ← autia($aret_b'$, $aret_i'$)
  str    LR,  [Xb, #r]     ; replace LR in jmp_buf
  b      <longjmp>
```

Listing 5: Before longjmp, the PACStack longjmp_wrapper[4] verifies the binding of the $aret_b'$, ret_b' and sp_b' values stored in jmp_buf. $\mathcal{A}$ cannot generate $aret_b'$ for arbitrary values and therefore cannot inject them in jmp_buf. #r, #a and #s are the offsets to ret_b, CR, and ret_i within jmp_buf.

5.4 Multi-threading

The values of ARMv8-A general purpose registers are stored in memory when entering EL1 (i.e. kernel-mode) from EL0 (i.e. user-mode), for example during context switches and system calls. This must not allow $\mathcal{A}$ to modify the $aret$ values or read the mask, which are both exclusively in either CR or LR during execution (Listings 2 and 3), but must be stored in memory during the context switch. On ARMv8-A, system calls are implemented using the supervisor call instruction (svc) that switches the CPU to EL1 and triggers a configured handler. On 64-bit ARM, Linux v5.0 uses the kernel_entry[5] macro to store all register values on the EL1 stack, where they cannot be accessed by user-space processes. During context switches, callee-saved registers (including CR) and LR are stored in struct cpu_context[6] which belongs to the in-kernel task structure and cannot be accessed by user space. The CR and LR values of a non-executing task are thus securely stored within the kernel, beyond the reach of other processes or other threads within the same process. Thus, no kernel modifications are needed to securely apply PACStack to multi-threaded applications.

6 Security evaluation

We address three questions in this section:
1) Is PAC reuse a realistic concern in prior PA-based schemes?
2) Is the ACS scheme cryptographically secure?
3) Do ACS's guarantees hold when instantiated as PACStack?

[4]Listings 4 and 5 are illustrative, complete wrapper code is available at https://github.com/pacstack/pacstack-wrappers

[5]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/entry.S?h=v5.0

[6]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/processor.h?h=v5.0

```
 1 void A() { stack_disclose(); }
 2 void B() {
 3     char buff[SIZE];
 4     stack_overflow(buff);
 5 }
 6 void func() {
 7     A(); // <---------------------------
 8     // ...    reusable return addresses |
 9     B(); // <---------------------------
10 }
```

Listing 6: The `-mbranch-protection` implementation (Section 2.2.1) computes the PAC for return addresses using the `SP` value at function entry. Both invocations in `func` (Lines 7 and 9) will thus use the same `SP` value as modifier. $\mathcal{A}$ can reuse the signed address from Line 7 to make the function invocation at Line 9 return to Line 8.

6.1 Reuse attacks on PA

Reuse attack on PA-based schemes are possible when the modifier is calculated with known or predictably repeating values. Using the `SP` can mitigate reuse attacks (Section 2.2.1). However, `-mbranch-protection` generates the PAC immediately on function entry, before modifying the `SP` value to allocate stack space. All functions called from within a code segment use the same modifier unless there are dynamic stack allocations. Moreover, because the stack is typically aligned to 8 bytes, the `SP` value will often repeat. For example, a less than $1s$ test execution of a SPEC CPU 2017 benchmark (`538.imagick_r`) already shows multiple collisions, with 5349 distinct (`LR`,`SP`) pairs, but only 914 unique `SP` values. Listing 6 shows a minimal example where all called functions will end up using the same modifier and thus have interchangeable signed return addresses.

6.2 ACS security

A generic representation of an attack against ACS is shown in Figure 4. Under normal operation, function C returns to A if called from A (Figure 4a); i.e., when called from A, the return address of C is an address ret_A in A. The goal of $\mathcal{A}$ (Figure 4b) is to cause C to return to some other address ret_B.

Since the authenticated return address $aret_A$ containing ret_A is protected from $\mathcal{A}$, in order to perform a backward-edge control-flow attack, $\mathcal{A}$ must achieve two goals successfully:

AG-Jump: Obtain an authenticated return address $aret_B$, valid with respect to some known modifier, which will validate successfully when C returns.

AG-Load: Violate the integrity of the call stack such that the `LR` register is loaded with $aret_B$ from AG-Jump rather than the correct authenticated return address $aret_A$.

This requires two returns: one from a 'loader' function to load $\mathcal{A}$'s $aret_B$ into `LR`, and another from C to the return address ret_B contained in $aret_B$.

In the analyses below, we treat the $auth$ token $\mathtt{H}_\mathtt{K}(\mathtt{P}, m)$ as a random oracle with respect to both the pointer P and modifier m. This means that if $\mathtt{H}_\mathtt{K}(P, m)$ has never been computed by a function call, $\mathtt{H}_\mathtt{K}(P, m)$ will match any value with probability 2^{-b}, independently of any other value $\mathtt{H}_\mathtt{K}(P', m')$. In the analysis below we assume that programs that share the same PA keys between multiple processes or threads employ the mitigation strategy against brute-force attacks described in Section 4.3. This assumption and the design of ACS ensure that there is no authentication oracle available: the only way to test whether an $auth$ token is valid with respect to some address and modifier is to attempt to return using the address and token, triggering a crash if the token is incorrect. The difficulty of achieving these goals therefore depends on whether $\mathcal{A}$'s desired control-flow violation follows the call graph of the program and whether $auth$ tokens are masked. Violating control-flow integrity while still traversing the call graph is easier because this allows $\mathcal{A}$ to harvest $auth$ tokens and search for collisions; violations that do not follow the call graph are more difficult because they require that $\mathcal{A}$ make one or more guesses, risking a crash.

6.2.1 Violations that follow the call graph

As $\mathcal{A}$ can harvest authenticated return pointers when they are written to the stack, the short $auth$ tokens mean that in the absence of masking an attacker can violate the integrity of the call stack by finding collisions in $\mathtt{H}_\mathtt{K}(\cdot, \cdot)$.

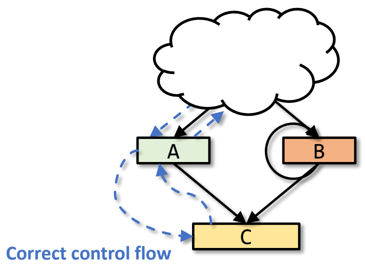
In order to achieve goal AG-Load, $\mathcal{A}$ must find two authenticated return addresses $aret_A$ and $aret_B$, such that i) they are both returned to by a function C, ii) that C contains a call-site to the loader function with a corresponding return address ret_C, and iii) such that

$$\mathtt{H}_\mathtt{K}(ret_C, aret_A) = \mathtt{H}_\mathtt{K}(ret_C, aret_B) = auth_{\text{collision}}. \tag{1}$$

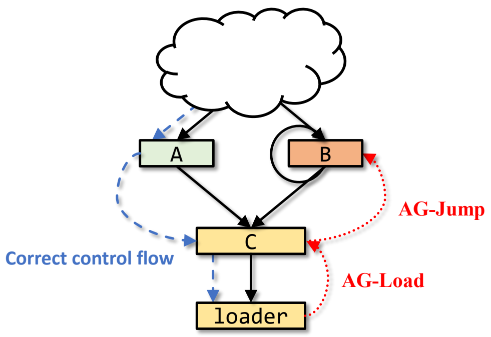
Note that the collisions must be for different values in the second argument only, since that is the value in $\mathcal{A}$'s control. Collisions that require different values for ret_C cannot be exploited because ret_C is in `CR` and cannot be modified by $\mathcal{A}$.

The $auth$ tokens contained in $aret_A$ and $aret_B$ depend on the path that $\mathcal{A}$ has taken through the call graph. $\mathcal{A}$ can obtain as many $auth$ tokens with ret_C as a pointer as there are distinct execution paths leading to C. The number of such paths will explode combinatorially as the complexity of the program increases, and cycles in the call graph—as occur in Figure 4—make the number of paths essentially infinite, limited only by available stack space.

Having found such a collision, $\mathcal{A}$ then arranges for function C to be called, traversing the call graph in such a way that it is set up to return to A using $aret_A$. Then, when the function C calls into the loader function, it will set `LR` to $aret_C$. When the



(a) Normal control flow.



(b) $\mathcal{A}$'s desired control flow.

Figure 4: Anatomy of a backward-edge control-flow attack against ACS. In order to force function C to return to B instead of its caller A, $\mathcal{A}$ substitutes their authenticated return address $aret_B$ when some function—the 'loader'—returns to ret_C in function C (goal AG-Load). If $aret_B$ is valid with respect to some known modifier, then at the end of function C the program will return to the corresponding ret_B (goal AG-Jump).

loader function returns to ret_C, it will attempt to load $aret_A$ from the stack. Instead, $\mathcal{A}$ substitutes $aret_B$, which because of (1) will validate correctly when returning to ret_C. Since $aret_B$ is a valid authenticated return address, C will successfully return to ret_B, thereby violating the integrity of the call stack.

More concretely, after collecting q *auth* tokens, according to the birthday paradox [47, Section 1.4.2], the probability that *some* pair collides is:

$$p_{\text{collision}}(q) = 1 - \frac{2^b!}{(2^b - q)! \cdot 2^{q \cdot b}}$$

This quickly approaches 1 as $\mathcal{A}$ collects more tokens, on average occurring after obtaining

$$q = \sqrt{\frac{\pi 2^b}{2}}$$

tokens. With a 16-bit PAC, $\mathcal{A}$ will therefore obtain a collision after harvesting 321 pointers on average.

In order to successfully mount the above attack, $\mathcal{A}$ must find two colliding *auth* tokens and perform the substitution. Without masking, $\mathcal{A}$ can read the *auth* token from the stack. $\mathcal{A}$ can then keep collecting *auth* tokens until they find two that collide; since these are both valid pointers, $\mathcal{A}$ will always succeed once this occurs, thus

$$\mathbb{P}[\text{AG-Load}|\text{Collision}] = 1.$$

With masking $\mathcal{A}$ cannot identify *auth* token collisions: $aret_A$ and $aret_B$ have different mask values $\mathtt{H}_\mathtt{K}(0, aret_A)$ and $\mathtt{H}_\mathtt{K}(0, aret_B)$. Therefore it is impossible to identify a collision with a probability greater than by random selection. This means that $\mathcal{A}$ will succeed in the attack above with a probability of 2^{-b}. We give a detailed proof in Appendix A.

In practice, this means that $\mathcal{A}$ can use this attack to traverse the program's call graph, but cannot jump to an address that is not a valid return address for function C.

Violation type	No masking	Masking
On-graph	1	2^{-b}
Off-graph to call-site	2^{-b}	2^{-b}
Off-graph to arbitrary address	2^{-2b}	2^{-2b}

Table 1: Maximum success probability of call-stack integrity violations, with and without masking.

6.2.2 Violations that leave the call graph

We now consider $\mathcal{A}$'s probability of success when attempting to return to an address ret_B in a way that *does not* follow the program's call graph. (Summary in Table 1.)

In this case, the path from B to C has not been traversed, and the instrumentation has never before computed the *auth* token $\mathtt{H}_\mathtt{K}(ret_C, aret_B)$. Therefore, $\mathcal{A}$ succeeds at AG-Load—i.e., $\mathtt{H}_\mathtt{K}(ret_C, aret_B) = \mathtt{H}_\mathtt{K}(ret_C, aret_A)$—with probability $\mathbb{P}[\text{AG-Load}] = 2^{-b}$, irrespective of whether the substituted $aret_B$ is a valid authenticated return address. On failure, which has probability $1 - 2^{-b}$, the process will crash.

$\mathcal{A}$'s probability of then achieving goal AG-Jump depends on whether ret_B is the return address of a valid call-site. If it is, then $\mathcal{A}$ can obtain a valid authenticated return pointer for that location in the same way as in Section 6.2.1. If ret_B has never been used as a return address, then no *auth* token has ever been generated for that pointer and AG-Jump is achieved with probability at most $\mathbb{P}[\text{AG-Jump}] = 2^{-b}$, independent of AG-Load.

$\mathcal{A}$ can therefore succeed with probability 2^{-b} when the return address is a valid call-site return address, or with probability of 2^{-2b} when the return address is not.

6.3 Run-time attack resistance of PACStack

PACStack must ensure the integrity of $aret_n$ and the confidentiality of the masks. The former is achieved by storing $aret_n$ in CR, which is reserved for this purpose, not used by regular code, and hence, inaccessible to $\mathcal{A}$ (Section 5.1). The latter is maintained as the mask is re-generated each time it is needed and cleared after use (Section 5.2). This holds true also in multi-threaded environments (Section 5.4). Traditional CFI solutions are unable to withstand control-flow bending [11]: attacks where each control-flow transfer follows the program's CFG, but the program execution trace conforms to no feasible benign execution trace. Schemes like PACStack and shadow call stacks are not susceptible to backward-edge control-flow bending because they precisely protect the integrity of the return addresses. $\mathcal{A}$ cannot trick PACStack to deviate from an expected return flow by replacing $aret_n$ with a valid, but outdated $aret$ value, because PACStack never writes $aret_n$ onto the stack. $\mathcal{A}$ also cannot reliably exploit PAC collisions to replace part of the $aret$ chain, as each $aret$ is masked. $\mathcal{A}$ cannot tamper with the instrumentation itself by modifying the instructions in memory (Assumption **A1**). By requiring coarse-grained forward-edge CFI (Assumption **A2**), PACStack ensures that $auth$ token calculations and masking are executed atomically and cannot be used to manipulate ret_i, $aret_{i-1}$ or the mask during the function prologue and epilogue. This holds when the forward-edge CFI is susceptible to control-flow bending (Section 3).

6.3.1 Tail calls and signing gadgets

A recent discovery by Google Project Zero [8] shows that PA schemes can be vulnerable to an attack whereby specific code sequences can be used as gadgets to generate PACs for arbitrary pointers. Recall that on PAC verification failure an `aut` instruction removes the PAC, but corrupts a well-known high-order bit such that the pointer becomes invalid. If a `pac` instruction adds a PAC to a pointer P with corrupt high-order bits, it treats the high-order bits *as though they were correct* when calculating the new PAC, and flips a well-known bit p of the PAC *if* any high-order bit was corrupt. This means that instruction sequences such as the one shown in Listing 7, consisting of an `aut` instruction followed by a `pac` instruction, can be used generate a valid PAC for a pointer even if the original pointer is not valid to begin with. $\mathcal{A}$ writes an arbitrary pointer P to memory (❶) and allows it to be verified. When verification fails, `autia` removes the PAC, and corrupts the high-order bit in P, writing the resulting P^* to the destination register (❷). The subsequent `pacia` will add the *correct PAC for* P, then flip bit p of the PAC to indicate that the input pointer was invalid (❸). $\mathcal{A}$ can now flip bit p back (❺) in order to obtain the correct PAC for pointer P (❻).

The PA signing gadget requires finding a matching ⟨`autia`, `pacia`⟩ pair operating on pointer P in the code without any use of P between these instructions. In PACStack each verification is immediately followed by a return, which ensures that the failure is detected. Tail calls are a notable exception. Tail calls are function calls executed before return and optimized so that the callee directly returns to the caller of the optimized function. For example, in Listing 8, function

```
...                   ; 𝒜 injects P at <ptr>                 ❶
ldr     Xd, <ptr> ; Xd ← P
autia   Xd, <mod> ; Xd ← P*                                   ❷
pacia   Xd, <mod> ; Xd ← pacia (P, <mod>) ⊕ p                ❸
str     Xd, <ptr> ; <ptr> ← Xd
...               ; 𝒜 sets <ptr> to <ptr> ⊕ p                ❺
ldr     Xd, <ptr> ; Xd ← pacia (P, <mod>)
autia   Xd, <mod> ; Xd ← P (valid pointer)                    ❻
```

Listing 7: A PAC is based on the address bits. An invalid input pointer (❶) after `aut` (❷) can be re-signed (❸), resulting in an output PAC with only a single bit-flip. This could be exploited to generate valid PACs for arbitrary pointers.

```
A:
  epilogue:
    ...
    ldr   X28, [SP]   ; load invalid aret'_{i-1}
    autia LR,  X28    ; LR ← ret*_i                 ③
    b     <B>         ; tail call B                 ①
B:
  prologue:
    str   X28, [SP]
    pacia LR,  X28    ; LR ← aret_i ⊕ p             ⑤
    ...
  epilogue:
    ...
    autia LR,  X28    ; LR ← ret*_i                 ④
    ret               ;                             ②
```

Listing 8: Tail calls on ARM replace the optimized call at the end of a function with a non-linking branch instruction (①).

`A` ends with a tail call to `B` using the `b` instructions that does not update `LR` (①). The tail-called function can return (②) to the `LR` value set before the tail call (③). PACStack limits $\mathcal{A}$ to modifying the previous $auth$ token on the stack. $\mathcal{A}$ could attempt to exploit the signing gadget to trick PACStack to accept an invalid $aret'_{i-1}$ (④), and subsequently load it into `LR` after return. However, $\mathcal{A}$ cannot flip the bit p of $aret'_i$ (⑤) because PACStack guarantees it is immutable. The invalid $aret'_{i-1}$ is thus always passed into `autia` (④) and so, detected at return from `B` (②). Forthcoming additions in the ARMv8.6-A architecture will preclude such attacks in general [3].

6.3.2 Sigreturn-oriented programming

Sigreturn-oriented programming [9] is a exploitation technique in UNIX-like operating systems, including Linux, that abuses the *signal frame* to take complete control of a process's execution state, i.e., the values of general purpose registers, `SP`, program counter (`PC`), status flags, etc. When the kernel delivers a signal, it suspends the process and changes the user-space processor context such that the appropriate signal handler is executed with the right arguments. When the signal handler returns, the original user-space processor context is restored. In a sigreturn attack $\mathcal{A}$ sets up a fake signal frame and initiates a return from a signal that the kernel never delivered. Specifically, a program returns from the handler using a `sigreturn` system call that reads a signal frame (`struct sigcontext` in Linux) from the process stack.

Although a sigreturn attack is, in principle, problematic for PACStack (as it could allow $\mathcal{A}$ control of any `EL0` register, including `CR`), a number of defenses against sigreturn attacks have been proposed for the Linux kernel, any of which will protect PACStack. Bosman and Bos [9] propose placing keyed signal canaries in the signal frame that are validated by the kernel before performing a `sigreturn`, or to keep a counter of the number of currently executing signal handlers. However, modern Linux versions rely solely on address space layout randomization (ASLR) [32] to make it difficult for the attacker to trigger an unwarranted `sigreturn`. Fortunately `sigreturn` is never called directly from program code (in fact the GNU C library `sigreturn` simply returns an error value). Instead the system call is triggered by signal trampoline code placed either in the kernel's virtual dynamic shared object (`vdso`) or in the C library, both subject to ASLR. For our chosen adversary model (Section 3) ASLR is not sufficient as $\mathcal{A}$ can determine the contents of any readable memory in the process memory space. However, PACStack itself, together with coarse-grained CFI (Assumption **A2**), ensures that $\mathcal{A}$ cannot divert control flow from program code to the signal trampoline. Nonetheless, 64-bit ARM programs that might call system calls directly using the `svc` instruction (without going through C library system call wrappers), would not be protected against the presence of such gadgets. We discuss a potential general solution against sigreturn attacks that utilizes the ACS construction in Appendix B.

7 Performance Evaluation

At present, the only publicly available PA-enabled SoCs are the Apple A12, A13, S4, and S5, none of which support PA for 3rd party code at the time of writing. To verify the correctness of instrumentation we ran all benchmarks on the ARMv8-A *Base Platform Fixed Virtual Platform (FVP)*, based on Fast Models 11.4, which supports ARMv8.3-A [2]. Because the FVP runs the v4.14 kernel, we have used PA RFC patches [7] modified to support all PA keys.

The FVP is not cycle-accurate and executes all instructions in one master cycle; therefore, it cannot be used for performance evaluation. Based on prior evaluations of the QARMA cipher [7], which is used as the underlying cryptographic primitive in reference implementations of PA [45], Liljestrand et al. estimate that the PAC calculations incur an average overhead of four cycles on a 1.2GHz CPU [35]. We employ the *PA-analogue* introduced by Liljestrand et al. to estimate the run-time overhead of PACStack.

[7] https://lwn.net/Articles/752116/

7.1 SPEC CPU 2017

We ran benchmarks on Amazon EC2 using the SPEC CPU 2017 benchmark package[8]. To guarantee exclusive access to the hardware, we used Amazon EC2 `a1.metal`[9] instances, each with 16 64-bit ARMv8.2-A cores. As these CPUs do not support PA, we instrumented benchmarks with the PA-analogue. For comparison, we measured run-time overheads of: 1) ShadowCallStack (a AArch64 production-ready software shadow call stack implementation for Clang 9 [14]), 2) `-mbranch-protection` (Clang's built-in PA-based return address protection), and 3) `-mstack-protector-strong` (stack canaries). We measured PACStack by instrumenting all function entry and exit points, excluding leaf functions that do not spill `LR` or the `CR` (this is similar to the heuristic used by `-mbranch-protection`). We measured both full PACStack and PACStack without masking (PACStack-nomask).

ShadowCallStack saves a function's return address in a separately-allocated shadow stack and then uses the protected return address when performing a return. On 64-bit ARM the `X18` register is reserved to hold a reference to the shadow stack. To perform a comparison against PACStack using the GNU C library (`glibc`) we ported ShadowCallStack support to `glibc`. Due to compatibility issues [52], we did not run the `perlbench` benchmarks with ShadowCallStack.

Our measurements include all C SPECrate and SPECspeed benchmarks, compiled with `-O2` optimizations and flags to enable the measured instrumentation. The suite is self-contained, avoiding the need to instrument system libraries. For each benchmark, we compared the performance of the baseline (with all evaluated instrumentations disabled) to the measured configuration. Figure 5 shows the mean overheads (w.r.t the baseline). Table 2 shows the geometric mean of the overheads, excluding `perlbench` which was incompatible with ShadowCallStack. On C++ benchmarks we observed overheads of 2.0% (PACStack) and 0.9% (PACStack-nomask). Due to compatibility issues with ShadowCallStack and `-mbranch-protection`, we limit our comparison to the C benchmarks.

As expected, `-mstack-protector-strong` outperforms other instrumentations (but provides the weakest protection). In terms of added instructions, `-mbranch-protection` is similar to PACStack-nomask; the performance difference is likely due to PACStack reserving the `CR` register and the additional store when saving it the stack. PACStack-nomask and ShadowCallStack have similar memory requirements (i.e., one extra store per function call), and show similar performance overheads. The overhead of PACStack is proportional to the frequency of function calls; benchmarks with few function calls are affected less than the benchmarks with frequent function calls. For instance, the `519.lbm_r` benchmark involves computations related to fluid dynamics and consists of large

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

[9] https://aws.amazon.com/ec2/instance-types/a1/

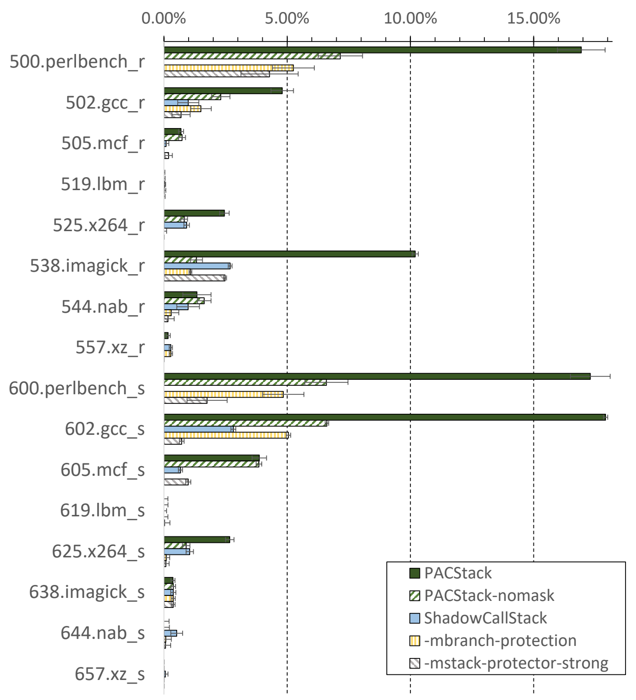


Figure 5: Relative performance overhead for SPEC CPU 2017 benchmarks as mean overhead over baseline. Error bars are 95% confidence intervals.

	SPECrate	SPECspeed
PACStack	2.75%	3.28%
PACStack-nomask	0.86%	1.56%
ShadowCallStack	0.85%	0.77%
`-mbranch-protection`	0.43%	0.72%
`-mstack-protector-strong`	0.43%	0.25%

Table 2: Geometric mean of measured overheads.

nested loops with few function calls. Consequently we see little effect on the performance of `519.lbm_r`.

Based on these results, we expect the overhead for both PACStack configurations to be a) comparable to ShadowCallStack, and b) negligible on PA-capable hardware.

7.2 Real-world evaluation: NGINX

We evaluated the efficacy of PACStack in a real-world setting using a SSL/TLS transactions per second (SSL TPS) test on the NGINX[10] open source web server software. SSL TPS measures a web server's capacity to create new SSL/TLS connections back to clients. Clients send a series of HTTPS requests, each on a new connection. The web server sends a 0-byte response to each request. The connection is closed after the response is received. We chose the SSL TPS test (instead of measuring throughput) to ensure that the load on the web server is CPU-bound, allowing us to estimate the upper bound for PACStack's impact on NGINX performance.

[10]https://www.nginx.com/

# of	Baseline		PACStack-nomask			PACStack		
workers	req./sec.	σ	req./sec.	σ	overhead	req./sec.	σ	overhead
4	14.2k	142	13.7k	124	3.8%	13.5k	117	5.5%
8	30.7k	722	28.6k	658	7.1%	27.2k	612	12.7%

Table 3: Requests/second, standard deviation (σ) and performance overhead for the NGINX SSL TPS tests reported for both PACStack and PACStack-nomask.

We conducted our tests on two separate Amazon EC2 A1 instances connected via elastic network interfaces with up to 10 Gbps capacity. The web server (on an `a1.metal` instance, running NGINX 1.17.8 with OpenSSL 1.1.1d) and the client (on an `a1.4xlarge` instance) ran the 64-bit ARM version of Ubuntu 18.04. We configured the server to use the ECDHE-RSA-AES256-GCM-SHA384 cipher with a 2,048-bit RSA key for HTTPS. The client used `wrk`[11], a modern HTTP benchmarking tool, to generate traffic. We configured `wrk` in the same way as in a test on NGINX performance conducted by F5 Networks.[12] We ran a total of 15 copies of `wrk` on the client machine for 3 minutes each.

We repeated the test with four and eight NGINX worker processes instrumented with PACStack and PACStack-nomask, and compared the results with uninstrumented baseline performance. In both configurations we also instrumented NGINX's dependencies (OpenSSL, `pcre` and `zlib` libraries). All binaries were compiled with `-O2` optimizations. We summarize the results in Table 3, showing a 4–7% overhead for PACStack-nomask and 6–13% overhead for PACStack. These results are consistent with the performance overheads measures for SPEC CPU 2017 (Section 7.1).

7.3 Compatibility testing using ConFIRM

ConFIRM is a set small micro-benchmarking suite designed to test compatibility and relevance of CFI solutions [52]. The suite is designed to test various corner-cases—e.g., function pointers, setjmp/longjmp and exception handling—that often cause compatibility issues for CFI solutions. ConFIRM is designed for x86-based architectures and includes some tests that are exclusive to the Microsoft Windows operating system. Of the 18 64-bit Linux tests 11 compiled and worked on AArch64; these included virtual and indirect function calls, setjmp/longjmp, calling conventions, tail calls and load-time dynamic linking. We ran these benchmarks on the FVP (to guarantee functional equivalence to PA-capable hardware) and confirmed that the tests passed with or without PACStack.

[11]https://github.com/wg/wrk (version of April 18, 2019)
[12]https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/

8 Related Work

Control-flow hijacking have been known for more than two decades [48]. Most current CFI solutions are *stateless*: they validate each control-flow transfer in isolation without distinguishing among different paths in the control-flow graph (CFG). *Fully-precise static CFI* [11] is the most restrictive stateless policy possible without breaking the intended functionality of the protected program. In fully-precise static CFI the best possible policy for return instructions is to allow returns within a function F to target any instruction that follows a call to F. All stateless CFI schemes, including fully-precise static CFI, are vulnerable to *control-flow bending* [11].

Stateful CFI can express policies that take previous control-flow transfers into account. *HAFIX* [19] is a hardware-assisted CFI scheme that confines function returns to active call sites. Context-sensitive CFI [20, 28, 51] further ensures that *each* control-flow transfer taken by the program is consistent with a non-malicious trace. Despite its better precision, context-sensitive CFI enforcement is considered impractical for real-world adoption [1]. Hardware-assisted branch recording features available in modern 64-bit Intel microprocessors can be used to enable context-sensitive CFI enforcement on commodity hardware, but suffer from i) limited branch history used to make CFI decisions, ii) over-approximation of the program CFG, and iii) reliance on complex run-time monitoring. *HAFIX*, on the other hand, requires changes to the processor.

As dynamic schemes, PACStack and *shadow call stacks* [1, 6, 13–15, 17, 18, 22–24, 29, 38, 39, 50] are not vulnerable to control-flow bending. Stateless forward-edge CFI enforcement is often combined with a shadow stack to enforce the integrity of return addresses stored on the call stack. In fact, the results by Carlini et al. [11] show that a shadow stack (or equivalent mechanism) is essential for the security of CFI. However, traditional shadow stacks incur significant performance overhead and lead to false positives for programming constructs that cause mismatches between calls and returns (C++ exceptions with stack unwinding, `setjmp`/`longjmp`). Recent designs improve performance by either leveraging a parallel shadow stack [17], or using a dedicated register for shadow stack addressing [10]. But since the shadow stack in this schemes resides in the same address space as the target application, it can be compromised if $\mathcal{A}$ knows its location. A typical solution for dealing with mismatches between calls and returns is to pop return addresses off the shadow stack until a match is found, or the shadow stack is empty (e.g., binary RAD [13]). This not only increases the complexity and run-time of the shadow stack instrumentation placed in the function epilogue, but also sacrifices precision, e.g., it allows $\mathcal{A}$ to redirect `longjmp` to any previously active call site. This can be avoided by storing and validating both the return address and stack pointer [15, 40, 50]. So far, only hardware-assisted shadow stacks promise to achieve negligible overhead without security trade-offs (e.g., Intel CET [29]).

Park et al. [42] present a micro-architectural shadow stack implementation using the branch predictor *return address stack*, a common hardware feature found in modern speculative superscalar processor designs. The return address stack is typically a circular buffer; to avoid losing stored return addresses when the maximum capacity is reached, Park et al. modify the return address stack to spill a portion of its content to backup storage in main memory. A Merkle-tree caching scheme is used to efficiently authenticate the backup storage before it is read back to the return address stack. The latency of spill/fill operations on backup memory is offset by the 100% hit rate for branch prediction since return addresses that exceed the return address stack capacity are retained.

The idea of using of MACs to protect the return address at run-time was introduced in *Cryptographic CFI* (CCFI) [37] which uses MACs to protect return addresses and other control-flow data (e.g., function pointers and C++ vtable pointers). CCFI's return address protection is similar to PA-based return address signing [45]; both bind the return address to the address of the function's stack frame and thus provide only coarse-grained resistance against pointer reuse [35]. In contrast to PACStack, these approaches cannot prevent reuse attacks (See Section 6.1). Independently to our work, Li et al. [34] propose a chain structure to protect return addresses but do not prevent the attacker from exploiting MAC collisions, and require custom hardware to realize their solution.

Program Counter Encoding [16, 22, 33, 41, 44] protects return addresses on the stack by encoding them with either a register-resident secret key [33], a read-only key stored in memory [16], the `SP` [44], or the address at which the return address itself is stored (a.k.a. the *self-address*) [41]. It is efficient, but relying on a secret key resident in user space makes such encoding schemes susceptible to buffer over-reads, and `SP` or self-address encoding suffer the same drawbacks as `-msign-return-address` [35, 45] (Section 2.2.1).

Other prominent defenses against control-flow attacks include fine-grained code randomization [32], and code-pointer integrity (CPI) [31]. Code randomization makes it more difficult for $\mathcal{A}$ to find suitable gadgets to exploit, but ineffective if $\mathcal{A}$ knows the program memory layout. CPI protects code pointers by storing them in a separate *safe stack*, which requires similar integrity guarantees as shadows stacks to remain effective [21]. Roessler et al. propose a metadata-tagged architecture to isolate stack-objects based on the stack-depth [46]. However, similar to the `SP` value (Section 6.2), the stack-depth will repeat frequently during program execution.

PACStack targets the ARM architecture, which has received less attention compared to the x86 family of computer architectures in terms of CFI research. *MoCFI* [18] is a software-based CFI approach specifically targeting ARM application processors used in smartphones. It uses a combination of a shadow stack, static analysis and run-time heuristics to determine the set of valid targets for control-flow transfers, but suffers from the same drawbacks that plague traditional shadow stack schemes. *CFI CaRE* [39] is a CFI solution targeting small, embedded ARM-based microcontrollers (MCUs). It uses the ability to perform hardware-enforced isolated execution on ARMv8-M MCUs to isolate the shadow stack to a secure processor state. The ARMv8-M [5] architecture enforces that calls to secure functions must target *secure gate instructions* placed at the beginning of such functions. The ARMv8.5-A architecture introduces similar *branch target indicators* (BTI) [4] to ARM application processors. BTI constitutes one way to meet the PACStack pre-requisite of coarse-grained CFI (Section 3).

9 Discussion

9.1 Support for software exceptions

The `setjmp` / `longjmp` interface has traditionally been used to provide exception-like functionality in C. However, modern coding standards for C and C++ that aim to facilitate code safety, security, and reliability consider them harmful and forbid their use, e.g., MISRA C:2004 [26, Rule 20.7] and JSF AV C++ [36, Rule 20]. Recall from Section 4.4 that calling `longjmp` with an expired `jmp_buf` is undefined behavior. For PACStack, this means that although the $aret_b$ in `jmp_buf` is tied to the corresponding SP and $auth_i$, its *freshness* cannot be guaranteed. $\mathcal{A}$ can modify `jmp_buf` to contain the previously used $aret_b$ and $\mathtt{SP}_b$, but must also modify the stack-frame at $\mathtt{SP}_b$, such that it contains the prior $aret_i$. This allows a control-flow transfer to a previously valid `setjmp` return site and SP value. To prevent reuse of expired `jmp_buf` buffers, `longjmp` can be rewound step-by-step, i.e., conceptually performing returns until the correct stack-frame is reached.

We plan to extend PACStack support to LLVM `libunwind`[13] – it does frame-by-frame unwinding of the call stack. By validating the ACS on each stack frame unwinding, PACStack can ensure that a fresh and valid state is reached.

As C++ exceptions also cause irregular stack unwinding they pose a similar challenge. But C++ does finer-grained stack unwinding to correctly destroy objects in unwound stack frames. The LLVM `libcxxabi` library will, depending on configuration, use `libunwind` for this purpose. With PACStack support in `libunwind`, we will be able to secure both `setjmp` / `longjmp` and support C++ exception handling.

9.2 Interoperability with unprotected code

Interoperability with unprotected (uninstrumented) code is an important deployment consideration. On one hand, PACStack-protected applications may need to interoperate with unprotected shared libraries. On the other, unprotected applications may need to interoperate with PACStack-protected shared libraries. The latter scenario is relevant for deployment in mobile operating systems like Android, where multiple stakeholders provide application binaries to consumer devices. The deployment of PACStack, or any other run-time protection mechanism, is likely to be driven by OEMs that enable specific protection schemes for the operating system and system applications. However, OEMs are not in control of native code deployed as part of applications. It should be possible for one version of the shared libraries shipped with the operating system to remain interoperable with both PACStack-protected, and unprotected apps.

In Section 5.1 we explain how the use of callee-saved registers allows PACStack to remain interoperable with unprotected code. Recall that because CR is a callee-saved register it will be restored upon return. However, PACStack cannot guarantee that CR remains unmodified during the execution of the unprotected code that could temporarily store its value on the stack. To meet the security guarantees (Section 6), PACStack instrumentation must be applied to both the application and any shared libraries. But partial protection, e.g. PACStack-protected shared libraries can significantly raise the bar for the attacker, as calls into protected functions can still benefit from return address authentication. Common shared libraries like `libc` are a popular source for gadgets for run-time attacks because of their size and availability. Because functions in a PACStack-protected library validate the return address in returns from library functions, they effectively remove a potentially large set of reusable gadgets from $\mathcal{A}$'s disposal.

10 Conclusion

ACS achieves security on-par with hardware-assisted shadow stacks (Section 6). With PACStack, we demonstrate how the general-purpose security PA security mechanism can realize our design, *without requiring additional hardware support or compromising security*. Other general-purpose primitives like memory tagging and branch target indicators are being rolled out. Creative uses of such primitives hold the promise of significantly improving software protection.

Acknowledgments

This work was supported in part by NSERC (RGPIN-2020-04744), Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), and Google (ASPIRE program). We acknowledge the computational resources provided by the Aalto Science-IT project.

References

[1] Martín Abadi et al. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[13]https://github.com/llvm/llvm-project/tree/master/libunwind

[2] ARM Ltd. Fast models version 11.4 reference manual. https://developer.arm.com/documentation/100964/1104-00/, 2018.

[3] ARM Ltd. Developments in the Arm A-profile architecture: Armv8.6-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a, 2019.

[4] ARM Ltd. ARM architecture reference manual (ARM DDI 0487F.c). https://developer.arm.com/documentation/ddi0487/fc, 2020.

[5] ARM Ltd. Armv8-M architecture reference manual (ARM DDI 0553B.l). https://developer.arm.com/documentation/ddi0553/bl/, 2020.

[6] Sergei Arnautov and Christof Fetzer. ControlFreak: Signature chaining to counter control flow attacks. In *Proc. IEEE SRDS '15*, pages 84–93, 2015.

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

[8] Brandon Azad. Google Project Zero: Examining pointer authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[9] Erik Bosman and Herbert Bos. Framing signals - a return to portable shellcode. In *Proc. IEEE S&P '14*, pages 243–258, 2014.

[10] Nathan Burow, Xingping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proc. IEEE S&P '19*, pages 985–999, 2019.

[11] Nicolas Carlini et al. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Security '15*, pages 161–176, 2015.

[12] Shuo Chen et al. Non-control-data attacks are realistic threats. In *Proc. USENIX Security '05*, pages 177–191, 2005.

[13] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. IEEE ICDCS '01*, pages 409–417, 2001.

[14] Clang 9.0 Documentation. ShadowCallStack. https://releases.llvm.org/9.0/tools/clang/docs/ShadowCallStack.html, 2019.

[15] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to protect return addresses from attack. *ARM SIGARCH Comput. Archit. News*, 33(1):65–72, 2005.

[16] Crispin Cowan et al. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security '03*, pages 91–104, 2003.

[17] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proc.ACM ASIA CCS '15*, pages 555–566, 2015.

[18] Lucas Davi et al. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc. NDSS '12*, 2012.

[19] Lucas Davi et al. HAFIX: Hardware-assisted flow integrity extension. In *Proc. ACM/EDAC/IEEE DAC '15*, pages 74:1–74:6, 2015.

[20] Ren Ding et al. Efficient protection of path-sensitive control security. In *Proc. USENIX Security '17*, pages 131–148, 2017.

[21] Isaac Evans et al. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. IEEE S&P '15*, pages 781–796, 2015.

[22] Michael Frantzen and Michael Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. USENIX Security '01*, pages 55–66, 2001.

[23] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proc. USENIX Security '02*, pages 61–79, 2002.

[24] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proc. NDSS '04*, 2004.

[25] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. Dynamic canary randomization for improved software security. In *Proc. ACM CISRC '16*, pages 9:1–9:7, 2016.

[26] HORIBA MIRA Ltd. Guidelines for the use of the C language in critical systems, 2004.

[27] Hong Hu et al. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proc. IEEE S&P '16*, pages 969–986, 2016.

[28] Hong Hu et al. Enforcing unique code target property for control-flow integrity. In *Proc. ACM CCS '15*, pages 1470–1486, 2018.

[29] Intel Corporation. Control-flow Enforcement Technology specification, revision 3.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2019.

[30] Tim Kornau. *Return Oriented Programming for the ARM Architecture*. PhD thesis, Ruhr-Universität Bochum, 2009.

[31] Volodymyr Kuznetsov et al. Code-pointer integrity. In *Proc. USENIX OSDI '14*, pages 147–163, 2014.

[32] Per Larsen et al. SoK: Automated software diversity. In *Proc. IEEE S&P '14*, pages 276–291, 2014.

[33] Gyungho Lee and Akhilesh Tyagi. Encoded program counter: Self-protection from buffer overflow attacks. In *Proc. CSREA ICIC '00*, pages 387–394, 2000.

[34] Jinfeng Li et al. Zipper stack: Shadow stacks without shadow. arXiv:1902.00888 [cs.CR], 2019.

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

[36] Lockheed Martin Corporation. Joint Strike Fighter Air Vehicle C++ Coding Standards (Revision C), 2005.

[37] Ali Jose Mashtizadeh et al. CCFI: Cryptographically enforced control flow integrity. In *Proc. ACM CCS '15*, pages 941–951, 2015.

[38] Danny Nebenzahl, Mooly Sagiv, and Avishai Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. *IEEE Trans. Dependable Secur. Comput.*, 3(1):78–90, 2006.

[39] Thomas Nyman et al. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proc. RAID '17*, pages 259–284. Springer International Publishing, 2017.

[40] H. Ozdoganoglu et al. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.*, 55(10):1271–1285, 2006.

[41] Seho Park, Yongsuk Lee, and Gyungho Lee. Program counter encoding for ARM® architecture. *Journal of Information Security*, 8:42–55, 2017.

[42] Yong-Joon Park and Gyungho Lee. Repairing return address stack for buffer overflow protection. In *Proc. ACM CF '04*, pages 335–342, 2004.

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

[44] Changwoo Pyo and Gyungho Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *Proc. ICICS '02*, pages 25–36, 2002.

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

[46] Nick Roessler and Andre DeHon. Protecting the stack with metadata policies and tagged hardware. In *Proc. IEEE S&P '18*, pages 478–495, 2018.

[47] Nigel P. Smart. *Cryptography Made Simple*. Springer Publishing Company, 1st edition, 2015.

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

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

[50] Caroline Tice et al. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. USENIX Security '14*, pages 941–955, 2014.

[51] Victor van der Veen et al. Practical Context-Sensitive CFI. In *Proc. ACM CCS '15*, pages 927–940, 2015.

[52] Xiaoyang Xu et al. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proc. USENIX Security '19*, pages 1805–1821, 2019.

A Security proofs

In Section 6.2, we gave an informal analysis of the security of ACS; here we give a more detailed proof of security, and in particular prove that authentication token masking prevents $\mathcal{A}$ from obtaining exploitable authentication token collisions.

The argument proceeds as follows: we suppose that $\mathcal{A}$, after obtaining q authentication tokens, can find a pair of inputs (x,y) and (x,y') whose authentication tokens $\mathtt{H}_\mathtt{K}(\cdot,\cdot)$ collide. This can be used to construct a distinguisher of the masks $\mathtt{H}_\mathtt{K}(0,\cdot)$ from a random string. The structure of the authentication tags is such that this further reduces to a semantic security game for one-time pad encryption of the masks. Then, we show that any violation of the integrity of an ACS-protected call stack also yields values whose authentication tokens collide as described above, allowing us to bound the probability of an integrity violation.

We summarize our notation in Table 4.

Games	
$G_{\textsf{ACS}}$ *(Figure 11)*	Security game for ACS integrity.
$G_{\textsf{PAC-Collision}}$ *(Figure 6)*	Security game for the identification of colliding authentication tokens.
$G_{\textsf{PAC-Distinguish}}$ *(Figure 7)*	Security game for the distinguishability of $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ from a random oracle.
G_1, G_2, G_3 *(Figure 8)*	Semantic security games for the mask $\mathtt{H}_{\mathtt{K}}(0,\cdot)$.

Adversary interfaces		
$G_{\textsf{ACS}}$	$\mathcal{A}_{\text{oracle-request}}$	Get path through the call-graph for which $\mathcal{A}$ wants the final authenticated return address pushed to the stack.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authenticated return address.
	$\mathcal{A}_{\text{ACS-Violation}}$	Return to the challenger authenticated return values that can be used to violate call stack integrity.
$G_{\textsf{PAC-Collision}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants a masked authentication token.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested masked authentication token.
	$\mathcal{A}_{\text{gen-collision}}$	Return to the challenger two authenticated return values with colliding authentication tokens.
$G_{\textsf{PAC-Distinguish}}$	$\mathcal{A}_{\text{oracle-request}}$	Get a value for which $\mathcal{A}$ wants an authentication tag.
	$\mathcal{A}_{\text{oracle-response}}$	Return a previously-requested authentication token.
	$\mathcal{A}_{\text{distinguish}}$	Return to the challenger a single bit identifying whether the given tokens were from a random oracle or $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$.
G_1, G_2	$\mathcal{B}_{\text{distinguish}}$	Identify the authentication token function used to generate masked authentication tokens.
G_3	$\mathcal{B}_{\text{distinguish'}}$	As for G_1, G_2, but with the inputs represented as strings, not functions.

Table 4: Notation used in Appendix A.

```
$G^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$
/ Give $\mathcal{A}$ $q$ masked authentication tokens
/ of their choice.
for $i \in \{1,\ldots,q\}$ do
    $(x,y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
    $\mathcal{A}_{\text{oracle-response}}(\mathtt{H}_{\mathtt{K}}(x,y) \oplus \mathtt{H}_{\mathtt{K}}(0,y))$
endfor
/ $\mathcal{A}$ is challenged to provide inputs whose authentication tokens collide.
$(\hat{x},\hat{y},\hat{y}') \leftarrow \mathcal{A}_{\text{gen-collision}}()$
if $\hat{y} \neq \hat{y}' \wedge \mathtt{H}_{\mathtt{K}}(\hat{x},\hat{y}) = \mathtt{H}_{\mathtt{K}}(\hat{x},\hat{y}')$ then
    return 1
endif
return 0
```

Figure 6: Security game for finding colliding PACs given masked authentication tokens.

```
$G^{\mathcal{A}}_{\textsf{PAC-Distinguish}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$

/ $\mathcal{B}$ is given values of their choice from either
/ $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ or a random oracle $RO(x,y)$
$S_0(x,y) \overset{def}{=} RO(x,y)$
$S_1(x,y) \overset{def}{=} \mathtt{H}_{\mathtt{K}}(x,y)$
$c \xleftarrow{\$} \{0,1\}$
for $i \in \{1,\ldots,q\}$ do
    $(x,y) \leftarrow \mathcal{A}_{\text{oracle-request}}()$
    $\mathcal{A}_{\text{oracle-response}}(S_c(x,y))$
endfor

/ $\mathcal{A}$ is challenged to determine whether it received
/ values from $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ or the random oracle.
$\hat{c} \leftarrow \mathcal{A}_{\text{distinguish}}()$
if $c \neq \hat{c}$ then
    return 1
endif
return 0
```

Figure 7: Security game in which $\mathcal{A}$ attempts to distinguish $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ from a random oracle.

Theorem 1 (PAC-masking prevents collision-finding)**.** *Suppose that after q queries, an adversary $\mathcal{A}$ can distinguish $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ from a random oracle with advantage no greater than $\mathsf{Adv}^{\mathcal{A}}_{\textsf{PAC-Distinguish}}(1^\lambda, H, q)$, as given in Figure 7. Then, assuming a key-length of λ for $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$, and given access to q masked authentication tokens, $\mathcal{A}$ can identify a pair of inputs $(\hat{x},\hat{y})$ and $(\hat{x},\hat{y}')$ whose corresponding unmasked authentication tokens collide with advantage at most $2\mathsf{Adv}^{\mathcal{A}}_{\textsf{PAC-Distinguish}}(1^\lambda, H, q)$.*

Proof. We begin with a collision-game $G^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q)$, shown in Figure 6 in which the adversary is given oracle access to the authentication token generator and then asked to provide values x, y, y' such that $\mathtt{H}_{\mathtt{K}}(x,y) = \mathtt{H}_{\mathtt{K}}(x,y')$.

An adversary that selects (x,y,y') at random from $\{0,1\}^{\texttt{VA\_SIZE}} \times \{0,1\}^{\texttt{VA\_SIZE}+b} \times \{0,1\}^{\texttt{VA\_SIZE}+b}$, such that $y \neq y'$, will win with probability 2^{-b}; $\mathcal{A}$'s advantage is therefore

$$\mathsf{Adv}^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q) = \mathbb{P}\left[G^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q) = 1\right] - 2^{-b}.$$

We will bound this advantage by reduction to a semantic security game for the masks. We consider the following games, shown in Figure 8, and described in Figure 9.

The first hop, from G_1 to G_2, is based on indistinguishability and relaxation: we suppose that $\mathtt{H}_{\mathtt{K}}(\cdot,\cdot)$ can be distinguished from a random oracle with probability no more than $\frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\textsf{PAC-Distinguish}}(1^\lambda, H, q)$, and that the adversary is not limited in the number of queries that can be made to the

$$\begin{tikzcd} \mathsf{G}_1^{\mathcal{B}}(1^\lambda, H, q) \arrow[r, bend left, "\mathtt{H_K}(\cdot,\cdot) \to \text{random oracle}"] & \mathsf{G}_2^{\mathcal{B}}(1^\lambda, H, q) \arrow[r, bend left, "\text{random oracle} \to \text{random string}"] & \mathsf{G}_3^{\mathcal{B}}(1^\lambda, H, q) \end{tikzcd}$$

```
$\mathsf{G}_1^{\mathcal{B}}(1^\lambda, H, q)$

$K \xleftarrow{\$} \{0,1\}^\lambda$
$S_0(y) \overset{def}{=} RO(y)$
$S_1(y) \overset{def}{=} \mathtt{H_K}(0,y)$
$T(x,y), x \neq 0$, first $q$ queries $\overset{def}{=} \mathtt{H_K}(x,y) \oplus \mathtt{H_K}(0,y)$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T(\cdot,\cdot)$.
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
    return 1
endif
return 0
```

```
$\mathsf{G}_2^{\mathcal{B}}(1^\lambda, H, q)$

$S_0(y) \overset{def}{=} RO_0(y)$
$S_1(y) \overset{def}{=} RO_1(0,y)$
$T(x,y), x \neq 0 \overset{def}{=} RO_1(x,y) \oplus RO_1(0,y)$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T(\cdot,\cdot)$.
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
    return 1
endif
return 0
```

```
$\mathsf{G}_3^{\mathcal{B}}(1^\lambda, H, q)$

$P_{1\ldots 2^{\mathtt{VA\_SIZE}}} \leftarrow \{0,\ldots,2^b - 1\}^{2^{b+\mathtt{VA\_SIZE}}}$
$S_0 \xleftarrow{\$} \{0,\ldots,2^b - 1\}^{2^{b+\mathtt{VA\_SIZE}}}$
$S_1 \xleftarrow{\$} \{0,\ldots,2^b - 1\}^{2^{b+\mathtt{VA\_SIZE}}}$
$T_{1\ldots 2^{\mathtt{VA\_SIZE}}} \leftarrow P_{1\ldots 2^{\mathtt{VA\_SIZE}}} \oplus S_1$

/ The adversary is given $S_0$ and $S_1$ and challenged to
/ determine which is used to calculate $T_{\ldots}$.
$c \xleftarrow{\$} \{0,1\}$
$\hat{c} \leftarrow \mathcal{B}_{\text{distinguish'}}(T, S_c, S_{1-c})$
if $c = \hat{c}$ then
    return 1
endif
return 0
```

Figure 8: Security games used in Theorem 1.

$G_1^{\mathcal{B}}(1^\lambda, H, q)$**:** $\mathcal{B}$ obtains masked authentication tokens $\mathtt{H_K}(x,y) \oplus \mathtt{H_K}(0,y)$ for up to q pairs (x,y) of $\mathcal{B}$'s choice, and must then distinguish the masks $\mathtt{H_K}(0,\cdot)$ from a random oracle.

$\downarrow$ $\mathtt{H_K}(\cdot,\cdot) \to$ random oracle

$G_2^{\mathcal{B}}(1^\lambda, H, q)$**:** This is the same as the previous game, except that $\mathtt{H_K}(\cdot,\cdot)$ is replaced by a random oracle and $\mathcal{B}$ is not limited in their number of queries. $\mathcal{B}$ must now distinguish between two random oracles, one of which is used in computing the authentication tokens, and one of which is independent of the authentication tokens.

$\downarrow$ Reformulation

$G_3^{\mathcal{B}}(1^\lambda, H, q)$**:** This is the semantic security game for repeated one-time-pad encryptions of a random string.

Figure 9: The game-hops used in Figure 8.

masked authentication token oracle. Then,

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda, H, q) = 1] \leq \quad \mathbb{P}[G_2^{\mathcal{A}}(1^\lambda, H, q) = 1] + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q).$$

The second hop, from G_2 to G_3, is a mere reformulation of G_2 such that random oracles are represented as strings, and that rather than allowing $\mathcal{B}$ to request arbitrarily many authentication tokens from the challenger, we instead give $\mathcal{B}$ direct access to the oracle, as represented by the sequence of strings $T_{1\ldots 2^{\mathtt{VA\_SIZE}}}$.

The third game is a semantic security game for the one-time pad, where $\mathcal{A}$ is given $2^{\mathtt{VA\_SIZE}}$ encryptions of S_1 and then asked to distinguish between S_1 and a random string. The perfect secrecy of the one-time pad means that $\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda) = 1] = \frac{1}{2}$ and so

$$\mathbb{P}[G_1^{\mathcal{B}}(1^\lambda) = 1] \leq \frac{1}{2} + \mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q). \quad (2)$$

Finally, we provide a reduction from $\mathsf{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$ to ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$. Suppose $\mathcal{A}$ can win $\mathsf{G}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$ with advantage $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$. Then, we define an adversary $\mathcal{A}^{\mathcal{A}}$ for ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$, shown in Figure 10.

This adversary wins ${\mathsf{G}_1}^{\mathcal{B}}(1^\lambda)$ with probability at least $\frac{1}{2} + \frac{1}{2}\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$, and so by (2)

$$\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q) \leq 2\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q).$$

If the MAC $\mathtt{H_K}(\cdot,\cdot)$ is a pseudo-random function family with respect to K, then $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Distinguish}}(1^\lambda, H, q)$ is negligible, and thus so is $\mathsf{Adv}^{\mathcal{A}}_{\text{PAC-Collision}}(1^\lambda, H, q)$. $\square$

```
$\mathcal{B}^{\mathcal{A}}_{\text{oracle-request}}()$

return $\mathcal{A}_{\text{oracle-request}}()$

$\mathcal{B}^{\mathcal{A}}_{\text{oracle-response}}(x)$

$\mathcal{A}_{\text{oracle-response}}(x)$

$\mathcal{B}^{\mathcal{A}}_{\text{distinguish}}(T, S, S')$

$x, y, y' \leftarrow \mathcal{A}_{\text{gen-collision}}(T)$
if $S(y) \oplus S(y') = T(x,y) \oplus T(x,y')$ then
    return 1
else
    return 0
endif
```

Figure 10: An adversary $\mathcal{B}^{\mathcal{A}}$ for G_1 used in our black-box reduction of $G_{\textsf{PAC-Collision}}$ to G_1. Not shown is the variant $\mathcal{B}^{\mathcal{A}}_{\text{distinguish'}}(T,S,S')$ that is identical to $\mathcal{B}^{\mathcal{A}}_{\text{distinguish}}(T,S,S')$ except that T, S, and S' are given in the form of strings.

With a bound on $\mathcal{A}$'s probability of successfully obtaining a PAC collision, we may now obtain a bound on their probability of violating the integrity of an ACS-protected call stack.

Theorem 2 (Security of ACS)**.** *Consider a program whose call stack is protected by ACS, which has a call-graph C and b-bit masked authentication tokens* $\mathtt{T}_{\mathtt{K}}(x,y) = \mathtt{H}_{\mathtt{K}}(x,y) \oplus \mathtt{H}_{\mathtt{K}}(0,y)$. *Then, an adversary with arbitrary control over memory can violate backward-edge control-flow integrity with probability*

$$\mathbb{P}\left[G^{\mathcal{A}}_{\textsf{ACS}}(1^\lambda, H, C, q)\right] \leq \mathbb{P}\left[G^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q)\right] \leq 2^{-b} + 2\mathit{Adv}^{\mathcal{A}}_{\textsf{PAC-Distinguish}}(1^\lambda, H, q)$$

Proof. We begin with a security game for ACS, shown in Figure 11.

Our goal is to provide a black-box reduction from $G^{\mathcal{A}}_{\textsf{ACS}}(1^\lambda, H, C, q)$ to $G^{\mathcal{A}}_{\textsf{PAC-Collision}}(1^\lambda, H, q)$.

From line 24 of Figure 11, winning $G^{\mathcal{A}}_{\textsf{ACS}}$ implies that $\mathcal{A}$ has obtained colliding authentication tokens, and therefore $\mathcal{A}$ can win $G^{\mathcal{A}}_{\textsf{PAC-Collision}}$ with probability at least $\mathbb{P}[G^{\mathcal{A}}_{\textsf{ACS}}]$. Substituting the bound from Theorem 1, we obtain the bound given. □

B Mitigation of sigreturn attacks

A solution for precluding sigreturn attacks against PACStack would be to include the *signal return value* to the PACStack chain via the PC value stored on the signal frame:

$$asigret_i = \begin{cases} \mathtt{H}_{\mathtt{K}}(sigret_i, asigret_{i-1}) & \text{if } i > 0 \\ \mathtt{H}_{\mathtt{K}}(sigret_i, aret_n) & \text{if } i = 0 \end{cases}$$

Upon signal delivery, the kernel stores a copy of *asigret*$_n$ securely in kernel space as a reference value. If the process

```
$G^{\mathcal{A}}_{\textsf{ACS}}(1^\lambda, H, C, q)$

 1:  $K \xleftarrow{\$} \{0,1\}^\lambda$
 2:  / Give $\mathcal{A}$ $q$ tokens from call-graph traversals.
 3:  for $i \in \{1, \ldots, q\}$ do
 4:      $p_{1\ldots m+1} \leftarrow \mathcal{A}_{\text{oracle-request}}()$
 5:      / Is the request for a real path through the call-graph?
 6:      if $\exists j : p_j \rightarrow p_{j+1} \notin \text{edges}(C)$ then
 7:          return 0
 8:      endif
 9:      $auth_m \leftarrow \mathtt{T}_{\mathtt{K}}(p_m, \mathtt{T}_{\mathtt{K}}(p_{m-1}, \cdots) \parallel p_{m-1}) \parallel p_m$
10:      $\mathcal{A}_{\text{oracle-response}}(auth_m)$
11:  endfor
12:  $ptr_{\textsf{jumper}}, ptr_{\textsf{correct}}, auth_{\textsf{correct}}, t_{\textsf{correct}},$
13:          $ptr_{\textsf{adv}}, auth_{\textsf{adv}}, t_{\textsf{adv}} \leftarrow \mathcal{A}_{\textsf{ACS-Violation}}()$
14:  / The substituted masked authenticated return address must be different.
15:  if $ptr_{\textsf{correct}} = ptr_{\textsf{adv}} \wedge auth_{\textsf{correct}} = auth_{\textsf{adv}}$ then
16:      return 0
17:  endif
18:  / Does the return pointer authenticate correctly with the adversary's
19:  / new masked authenticated return address as the modifier?
20:  if $\mathtt{H}_{\mathtt{K}}(ptr_{\textsf{jumper}}, auth_{\textsf{correct}} \parallel ptr_{\textsf{correct}})$
21:      $\neq \mathtt{H}_{\mathtt{K}}(ptr_{\textsf{jumper}}, auth_{\textsf{adv}} \parallel ptr_{\textsf{adv}})$ then
22:      return 0
23:  endif
24:  / Did the adversary provide a valid masked authenticated return address?
25:  if $auth_{\textsf{adv}} = \mathtt{H}_{\mathtt{K}}(ptr_{\textsf{adv}}, t_{\textsf{adv}})$
26:      return 1
27:  endif
28:  return 0
```

Figure 11: Security game for ACS with respect to a program having call-graph C and authentication token function $\mathtt{T}_{\mathtt{K}}(\cdot,\cdot)$.

was already executing a signal handler, and thus the kernel already has a reference copy of *asigret*$_{n-1}$ on record, it stores *asigret*$_{n-1}$ in the new signal frame and overwrites the secure copy with *asigret*$_n$. On `sigreturn` the kernel attempts to validate the PC and CR values in the signal frame as though the reference value was *asigret*$_0$. If successful it performs the signal return to *sigret*$_n$ and restores *aret*$_n$ to CR. Otherwise the kernel assumes a return to a nested signal handler, and retrieves $sigret'_n$ and $asigret'_{n-1}$ from the signal frame, validates them by calculating $asigret'_n = \mathtt{H}_{\mathtt{K}}(sigret'_n, asigret'_{n-1})$ and comparing the result against the stored *asigret*$_n$ reference value. If successful the kernel replaces *asigret*$_n$ with *asigret*$_{n-1}$ in the secure kernel store and performs the signal return to *sigret*$_n$. If the validation fails the kernel terminates the process. This prevents $\mathcal{A}$ from 1) overwriting CR, and 2) forging the PC values in signal frames. For general protection against sigreturn attacks corrupting any register stored in the signal frame, all register values could be included in the *asigret* calculation using the `pacga` instruction and validated at the time of `sigreturn`.