Tightly Seal Your Sensitive Pointers with PACTIGHT

Mohannad Ismail *Andrew Quach* † *Christopher Jelesnianski* *Yeongjin Jang* † *Changwoo Min*
Virginia Tech † *Oregon State University*

Abstract

ARM is becoming more popular in desktops and data centers, opening a new realm in terms of security attacks against ARM. ARM has released *Pointer Authentication*, a new hardware security feature that is intended to ensure pointer integrity with cryptographic primitives.

In this paper, we utilize Pointer Authentication (PA) to build a novel scheme to completely prevent any misuse of security-sensitive pointers. We propose PACTIGHT to tightly seal these pointers. PACTIGHT utilizes a strong and unique modifier that addresses the current issues with the state-of-the-art PA defense mechanisms. We implement four defenses based on the PACTIGHT mechanism. Our security and performance evaluation results show that PACTIGHT defenses are more efficient and secure. Using real PA instructions, we evaluated PACTIGHT on 30 different applications, including NGINX web server, with an average performance overhead of 4.07% even when enforcing our strongest defense. PACTIGHT demonstrates its effectiveness and efficiency with real PA instructions on real hardware.

1 Introduction

In recent years, the ARM processor architcture started penetrating into the data center [8, 52, 53] and mainstream desktop [10] markets beyond the mobile/embedded segments. This opens a new realm in terms of security attacks against ARM, increasing the importance of having effective and efficient defense mechanisms for ARM.

Control-flow hijacking attacks are one of the most critical security attacks. These attacks aim to subvert the control-flow of a program by carefully corrupting code pointers, such as return addresses and function pointers. Control-flow integrity (CFI) [6] aims to defend against these attacks by ensuring that the program follows its proper control-flow. This is mainly done by generating a control-flow graph (CFG) of the program and making the program conform to it.

In order to defend against control-flow hijacking attacks efficiently, ARM has introduced a new hardware security feature, *Pointer Authentication (PA)* [34], which ensures pointer integrity with cryptographic primitives. PA computes a cryptographic MAC called a *Pointer Authentication Code (PAC)* and stores it in the unused upper bits of a 64-bit pointer. PA can be used to defend against control-flow hijacking attacks securely and efficiently with low performance and memory overhead.

However, PA is not almighty. Although several PA-based defense mechanisms have been proposed [27, 41–43] and deployed [11, 34], we identified that they are still exposed to attacks, such as using a signing gadget to forge PACs [15] and reusing PACs [43], allowing arbitrary code execution.

In this paper, we propose PACTIGHT, which is a PA-based defense against control-flow hijacking attacks. In particular, we define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with. They are: 1) *unforgeability*: A pointer `p` should always point to its legitimate object; 2) *non-copyability*: A pointer `p` can only be used when it is at its specific legitimate location; 3) *non-dangling*: A pointer `p` cannot be used after it has been freed. PACTIGHT tightly seals pointers and guarantees that a sealed pointer cannot be forged, copied, or dangling.

Compared to previous PA-based defense mechanisms, PACTIGHT assumes a stronger threat model such that an attacker has both arbitrary read and write capabilities. PACTIGHT also provides better coverage by protecting a variety of security-sensitive pointers. In this paper, we define a sensitive pointer as any pointer that can reach a code pointer. PACTIGHT enforces the three properties in order to prevent the pointers from being abused. Enforcement of the three properties protects against attacks that rely on manipulating the pointers.

We design PACTIGHT to achieve pointer integrity by protecting all sensitive pointers and by providing spatial and temporal memory safety for those sensitive pointers. Protecting these sensitive pointers achieves the balance between full memory safety and covering only control-flow hijacking. This allows for reinforced protection, thus achieving protection against control-flow hijacking attacks and providing memory safety for sensitive pointers. We demonstrate the effectiveness and practicality of PACTIGHT by evaluating with real PA instructions on real hardware.

In summary, we make the following contributions:

- We propose PACTIGHT, a novel and efficient approach to tightly seal pointers using PAC. By utilizing PACTIGHT's mechanisms, we make pointers unforge-

able, non-copyable, and non-dangling.

- We implemented four defenses using PACTIGHT: forward-edge protection, backward-edge protection, C++ VTable pointer protection, and all sensitive pointer protection.
- We provide a strong security evaluation by demonstrating effectiveness against real-world CVEs and synthesised attacks.
- We evaluate PACTIGHT implementations on SPEC CPU2006, nbench, CoreMark benchmarks, and NGINX web server with real PAC instructions. We show that PACTIGHT implementations achieve low performance and memory overhead, 4.07% and 23.2% respectively making it possible to deploy PACTIGHT defenses in the real-world.

2 Background and Motivation

In this section, we introduce control-flow hijacking attacks and ARM's pointer authentication mechanism. We then discuss defenses based on PAC and their limitations to motivate our work.

2.1 Control-Flow Hijacking Attacks

Control-flow hijacking attacks are critical attacks to computer systems because they may allow attackers to run arbitrary code on the system. A popular way to carry out a control-flow hijacking attack is to exploit memory corruption vulnerabilities, which C/C++ programs are prone to having. In particular, attackers can alter the value of a code pointer (*e.g.*, return addresses and function pointers) by corrupting the memory location that stores the pointer to subvert the execution flow of a program [16, 17, 20, 22, 26, 30, 59].

To defeat the attack, defenders must ensure that the program has no single point that can let an attacker corrupt code pointers as well as data pointers that refer to code pointers in its recursive memory dereference chain. Return-oriented programming (ROP) [56], jump-oriented programming (JOP) [17], and counterfeit-object oriented programming (COOP) [54] are the techniques that aim to achieve code execution by chaining returns, indirect call/jumps, and virtual function calls in an object iteration loop, respectively.

2.2 ARM Pointer Authentication

ARMv8.3-A [34] introduced a new hardware security feature, Pointer Authentication (PA). PA has been implemented in the Apple A12 and M1 chips [61]. The goal of PA is to protect the integrity of security-critical pointers, such as code pointers. To this end, a pointer authentication code (PAC) is generated by a cryptographic hash function, as a message authentication code (MAC), to put cryptographic integrity protection on the pointers. A PAC is a MAC of the target pointer value, a secret key, and a salt, which is a 64-bit modifier. The modifier can be tweaked to bind the context of the program when generating a PAC for a pointer. Some examples of such context are conveying the type of the pointer as a modifier, using stack frame address as a modifier, etc.

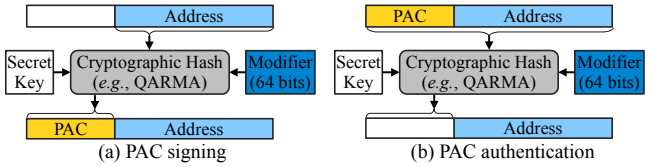

Figure 1: PA signs a pointer and generates a pointer authentication code (PAC) based on a address, a secret key, a 64-bit user-provided modifier using PA instructions (*e.g.*, `pacia`). The signed pointer should be authenticated before the access using the same PAC, address, secret key, and modifier using PA instructions (*e.g.*, `autia`).

PAC signing. PAC utilizes a cryptographic hash algorithm, namely QARMA [14]. The algorithm takes two 64-bit values (pointer and modifier), as well as a 128-bit key, and generates a 64-bit PAC. These PACs are truncated and added to the upper unused bits of the 64-bit pointer as illustrated in Figure 1(a). Five keys in total can be chosen to generate the PACs. These keys are stored in special hardware registers protected by the kernel.

PAC authentication. The cryptographic algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer. To pass the authentication, both values need to be the same as the ones originally used to generate the PAC. If the regenerated PAC matches, the PAC is removed from the pointer and the pointer can be used, as shown in Figure 1(b). Otherwise, the top two bits of the pointer are flipped, rendering the pointer unusable. Any use of the pointer results in a segfault.

PAC instructions. PAC instructions start with either `pac` or `aut` followed by a character that identifies whether it protects a code pointer, data pointer or generates a generic PAC. This is then followed by another character that identifies which key is being used. For example, the `pacib` instruction generates a PAC for a code pointer that uses the B-key. When authenticating this code pointer, the authenticate instruction for the code pointer and B-key, *i.e.*, `autib`, must be used to successfully authenticate. Without this, the pointer cannot be used as its semantics are changed.

Defense	Protection scope	Attacker abilities	PAC modifier	Unforegablity	Non-copyability	Non-dangling
PARTS-CFI [43]	Return addresses and indirect code pointers	Arbitrary read-write	SP for return addresses and type-id for indirect code pointers.	✓	×	×
PACStack [42]	Return addresses	Arbitrary read-write	Previous return address on the stack	✓	✓	×
PTAuth [27]	Heap allocated objects	Arbitrary write	A generated object-id	✓	×	✓
PACTIGHT	All sensitive pointers and return addresses	Arbitrary read-write	The location of the pointer and a random tag for sensitive pointers. Previous return address and a unique function id for return addresses.	✓	✓	✓

Table 1: Comparison between PACTIGHT and state-of-the-art PAC-based defense mechanisms.

2.3 PAC Defense Approaches

Return address focused. Qualcomm's return address signing mechanism [34] protects return addresses from stack memory corruption. It utilizes the `paciasp` and `autiasp` instructions. These are specialized instructions that sign the return address in the Link Register (LR) using the Stack Pointer (SP) as the modifier and the A-key to protect return addresses.

However, because this approach is susceptible to PAC re-use attacks (see §2.4), PARTS return address protection [43] includes the SP with a function ID as a modifier to harden the PAC scheme against re-use attacks. Moreover, PACStack [42] extends the modifier by chaining PACs to bind all previous return addresses in a call stack. On the other hand, PCan [41] relies on protecting the stack with canaries generated with PAC using a modifier consisting of a function ID and the least-significant 48 bits from SP.

Other code pointers. Apple extended its protection to cover other pointer types including function pointers and C++ VTable pointers. However, it uses a zero modifier to protect them. PARTS [43] utilizes PAC to protect function pointers, return addresses, and data pointers. It utilizes a type ID based on *LLVM ElementType* as the modifier for signing function pointers and data pointers.

Temporal safety. PTAuth [27] enforces temporal memory safety using PAC. PTAuth generates a new random ID at each memory allocation and utilizes it as a modifier for generating a PAC. Because the corresponding random ID of a pointer is cleared or updated when the pointer is being freed or allocated, PTAuth detects the violations of temporal memory safety (*e.g.*, use-after-free) by maintaining it as a modifier to check the liveness of a pointer at the time of authentication.

2.4 Limitations of Current PAC Defenses

Forging PAC. PAC relies on the security of the cryptographic hash, that is, attackers cannot generate a valid PAC for a pointer, even if they have both the pointer address value and the corresponding modifier. However, a memory corruption vulnerability in PAC generation logic may serve as an arbitrary PAC generator, allowing attackers to bypass the PAC authentication [15].

Reusing valid PACs in a different context. A PAC generated for one context can be reused in a different context if two contexts share the same modifier. This applies not only to the case of using zero modifier, such as Apple's virtual function table protection, but also to the case that shares the same modifier across different contexts, such as Qualcomm/Apple stack protection. An example case of the latter is to reuse the PAC generated for a valid return address with a specific SP at a different return location that shared the same SP (*e.g.*, having multiple function calls in a function, a case for sharing the same stack frame for all of its returns). PARTS-CFI is also susceptible to this attack because the approach uses a static modifier for the pointer, based on its LLVM ElementType. Having two different pointers of the same type, such two pointers will share the same modifier, and in such a case, attackers can reuse the PAC generated for one in the context of using the other. Table 1 summarizes the comparison between PACTIGHT and other existing state-of-the-art PAC defense mechanisms.

Reusing dangling PACs. Attackers can reuse legitimately generated PACs, even after a pointer becomes dangling. This occurs if the modifier used for signing the pointer does not convey the temporal state of the pointer. In such a case, the PAC is still valid even after deallocation of the memory referred to by the pointer, and thereby, attackers may reuse a valid PAC for a different object that the PAC has signed. In particular, there is no mechanism in PARTS or Apple's Clang to dynamically check and confirm if the pointers that they protect are not *dangling*, thus they are susceptible to this attack.

3 Threat Model and Assumptions

Our threat model assumes a powerful adversary with read and write capabilities by exploiting input-controlled memory corruption errors in the program. The attacker cannot inject or modify code due to Data Execution Prevention (DEP), which is by default enabled in most modern operating systems [35, 49]. Also, the attacker does not control higher privilege levels. We assume that the hardware and kernel are trusted, specifically that the PA secret keys are generated, managed and stored securely. Attacks targeting the kernel and hardware, such as Spectre [37], and data only attacks, which modify and

leak non-control data, are out of scope. Our assumptions are consistent with prior works [23, 41–43] with the exception of PTAuth [27], which only allows arbitrary write and not arbitrary read.

4 PACTIGHT Design

In this section, we describe the design of PACTIGHT. We first discuss our design goal (§4.1), then we introduce three pointer integrity properties that PACTIGHT enforces to overcome the limitations of prior PAC approaches (§4.2), and then we compare PACTIGHT to current state-of-the-art defenses (§4.3). Lastly, we present the detailed design of PACTIGHT. As shown in Figure 2, PACTIGHT consists of a runtime library and compiler-based instrumentation. We first discuss the runtime (§4.4) to explain how PACTIGHT enforces the pointer integrity properties and then explain PACTIGHT's automatic instrumentation and defense mechanisms (§5).

4.1 PACTIGHT Design Goals

The overarching goal of PACTIGHT is to completely prevent control-flow hijacking attacks in a program with low performance overhead. While prior works on PAC show promising results, they are limited in scope and/or security protection as discussed in §2.3. To achieve our goal, it is essential to enforce the complete integrity of pointers, which we will discuss in §4.2, and prevent any pointer misuse. We protect sensitive pointers [38] – all code pointers and all data pointers that are reachable to any code pointer – because guaranteeing the integrity of all sensitive pointers is sufficient to make control-flow hijacking impossible. In summary, our main goals are:

- **Integrity**: Prevent any misuse of sensitive pointers.
- **Performance**: Minimize runtime performance and memory overhead.
- **Compatibility**: Allow protection of legacy (C/C++) programs without any modification.

4.2 PACTIGHT Pointer Integrity Property

Based on the limitations of prior PAC approaches and our observation on how a pointer can be compromised, we define three security properties of pointer integrity, discussed in detail below:

- **Unforgeability**: As illustrated in Figure 3(a), a pointer can be forged (*i.e.*, corrupted) to point to an unintended memory object. Many memory corruption-based control flow hijacking attacks fall into this category by directly corrupting pointers (*e.g.*, indirect call, return address). With the *unforgeability* property, a pointer always points to its legitimate memory object and it cannot be altered maliciously.
- **Non-copyability**: A pointer can be copied and re-used maliciously as illustrated in Figure 3(b). Many information leakage-based control flow hijacking attacks first collect live code pointers and reuse the collected live pointer by copying them to subvert control flow. With the *non-copyability* property, a pointer cannot be copied maliciously. It asserts that a live pointer can only be referred from its correct location, preventing the re-use of live pointers at different sites. If *non-copyability* is guaranteed, the security impact is *non-replayability*, and thus pointer attacks that replay PAC-ed pointers for malicious use are prevented.
- **Non-dangling**: A pointer can refer to an unintended memory object if its pointee object is freed or the freed memory is reallocated as shown in Figure 3(c). The integrity of a pointer is compromised even if the pointer itself is not directly forged or copied. Semantically, the life cycle of a pointer should end when its pointee object is destructed. Many attacks exploiting temporal memory safety violation reuse such dangling pointers. With the *non-dangling* property, a pointer cannot be re-used after its pointee object is freed.

The importance of these properties stems from the fact that to hijack control-flow, at least one of these properties must be violated. PACTIGHT is able to detect any of these violations before the use of a pointer, thus guaranteeing the above mentioned pointer integrity. Note that ARM PAC only enforces the unforgeability property.

4.3 Comparison against Other PAC-based Defenses

In contrast to other PAC-based defenses (Table 1), PACTIGHT offers more coverage against PAC attacks. PARTS [43] relies on a static modifier based on the *LLVM ElementType*, which can be repeated. Even though an attack based on this would be harder than when using the `SP` as a modifier, it is still possible. PACTIGHT's unique modifier scheme eliminates any *replayability* of PACs, and thus defends against PAC reuse.

PACStack [42] introduces the idea of cryptographically binding a return address to a particular control-flow path by having all previous return addresses in the call stack influence the PA modifier. PACStack only protects return addresses on the stack and needs a forward-edge CFI scheme with it, whilst PACTIGHT protects all sensitive pointers on the stack and elsewhere.

PTAuth [27] attempts to provide protection against

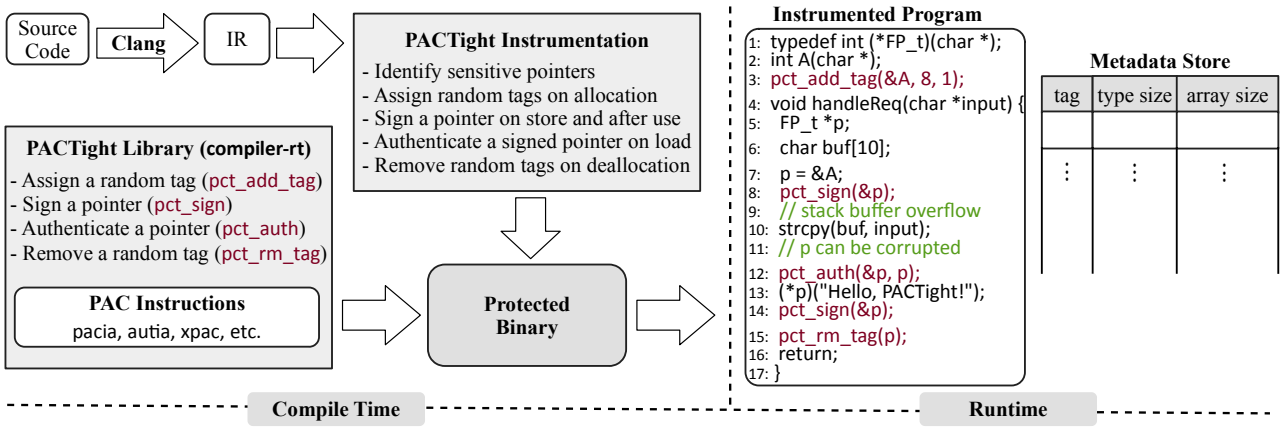

Figure 2: PACTIGHT design. At compile time, PACTIGHT instruments the allocation, assignment, use, and deallocation of code pointers and data pointers that are reachable to a code pointer (*i.e.*, sensitive pointers). PACTIGHT guarantees three pointer integrity properties (§4.2), namely unforgeability, non-copyability, and non-dangling. At runtime, PACTIGHT generates a PAC for sensitive pointers using a novel authentication scheme and checks the PAC upon pointer dereference (§4.4). PACTIGHT automates its instrumentation in four different levels: forward edge, backward edge, C++ VTable, and sensitive pointers (§5).

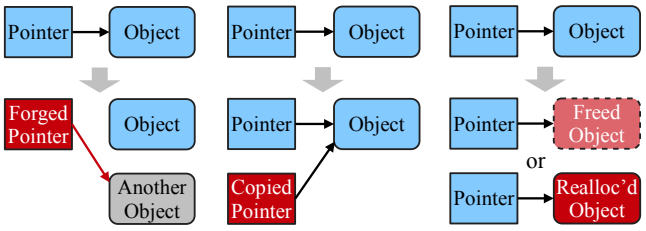

Figure 3: Three types of violations of pointer integrity.

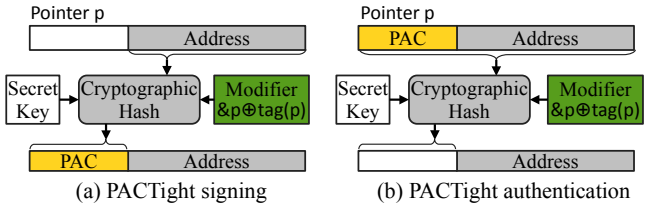

Figure 4: Signing and authentication of a pointer variable `p` in PACTIGHT. In addition to the unforgeability of `p` provided by PA, PACTIGHT uses the address of a pointer (`&p`) and a random tag associated with a pointee (`tag(p)`) to provide the non-copyability and non-dangling properties.

temporal attacks. However, it assumes a weaker threat model, defending against attackers with arbitrary write only. Also, it is vulnerable to intra-object violation. If two pointers within the same object are swapped, PTAuth cannot detect this. Thus, the pointers are *copyable* in this case. Moreover, it only protects the heap and does not handle stack protection, even from temporal attacks. PACTIGHT defends against a strong attacker, with arbitrary read and write capabilities, protects the stack, heap, global variables, and defends against any *forging*, *copyability*, and *dangling* of pointers.

4.4 PACTIGHT Runtime

This section describes the PACTIGHT runtime. We first describe how PACTIGHT efficiently enforces the pointer integrity properties (§4.4.1), then discuss the PACTIGHT runtime library (§4.4.2), pointer operations (§4.4.3) and the metadata store design (§4.4.4).

4.4.1 Enforcing PACTIGHT Pointer Integrity

In order to enforce the three properties, PACTIGHT relies on the PAC modifier. The modifier is a user-defined salt that is incorporated by the cryptographic hash into the PAC in addition to the address. Any changes in either the modifier or the address result in a different PAC, detecting the violation. We propose to blend the address of a pointer (`&p`) and a random tag (`tag(p)`) associated with a memory object to efficiently enforce the PACTIGHT pointer integrity property, as illustrated in Figure 4.

- **Unforgeability**: PAC by itself enforces the unforgeability of a pointer. PAC includes the pointer as one of the inputs to generate the PAC. If the pointer is forged, it will be detected at authentication.
- **Non-copyability**: PACTIGHT adds the location of the pointer (`&p`) as a part of the modifier. This guarantees that the pointer can only be used at that specific location. Any change in the location by copying the pointer (*e.g.*, `q = p`) changes the modifier (`&q`) and thus triggers an authentication fault.
- **Non-dangling**: PACTIGHT uses a random tag ID to track the life cycle of a memory object. PACTIGHT assigns a 64-bit random tag ID to a memory object upon allocation and deletes it upon deallocation. This is done for both stack and heap allocations. A random tag ID of a memory object (`tag(p)`) is blended with the location of the pointer (`&p`) to get the 64-bit modifier

for PAC generation and authentication. This implies that the life cycle of a PACTIGHT-sealed pointer is bonded to that of a memory object. When memory is deallocated (or re-allocated), PACTIGHT deletes (or re-generates) the random tag. This invalidates all pointers to that memory, enforcing the non-dangling property.

By incorporating all these pieces of information (*i.e.*, `p`, `&p`, and `tag(p)`) together into the PAC, PACTIGHT effectively enforces the three security properties for pointer integrity. Any change to any of the information results in a PAC authentication failure. Note that we used XOR to blend the location of a pointer and pointee's random tag into a single 64-bit integer.

4.4.2 Runtime Library

The PACTIGHT runtime library provides four APIs to enforce pointer integrity. The PACTIGHT LLVM instrumentation passes described in §5 automatically instrument a program using those APIs. The code for this library is presented in §A.1.

1) `pct_add_tag(p,tsz,asz)` sets the metadata for a newly allocated memory region. Besides a pointer `p`, it takes two additional arguments – the size of an array element (`tsz`) and the number of elements in the array (`asz`) in order to support an array of pointers. The PACTIGHT runtime assigns the same random tag for each array element. For each element, its associated random tag and size information are added to the metadata store. This means that each array element's metadata can be looked up separately. The API should be called whenever memory is allocated (heap or stack). PACTIGHT assigns a random tag to an object right after its allocation.

2) `pct_sign(&p)` signs a pointer with the associated random tag that was generated by `pct_add_tag`. It generates a 64-bit modifier using the location of a pointer (`&p`) and its associated random tag (`tag(p)`) by looking up the metadata store. Then, it signs the pointer with the modifier using a PA signing instruction (*e.g.*, `pacia`, `pacda`). If a (compromised) program tries to sign a pointer that does not have an associated random tag (*i.e.*, the program tries to access unallocated memory as in a use-after-free vulnerability), PACTIGHT aborts the program. This API should be called whenever a pointer is assigned or after it is used.

3) `pct_auth(&p,p+N)` authenticates a pointer with the associated metadata. Similar to `pct_sign`, it generates the modifier using the pointer location (`&p`) and its associated random tag (`tag(p+N)`) by looking up the metadata store, where `N` is the array index. `N` is zero in cases other than arrays. The use of `p+N` allows support for pointer arithmetic and enforcing spatial safety, which will be explained with an example in §4.4.3 (see Figure 5). Then, it authenticates the pointer with the modifier using a PA authentication instruction (*e.g.*, `autia`, `autda`). If there is no random tag or PA fails authentication, PACTIGHT aborts the program. Any value of `N` that is not within the bounds of the array will not return the correct tag, and thus also causes a failed authentication. If the authentication is successful, it strips off the PAC from the pointer. This API should be called before using the pointer.

4) `pct_rm_tag(p)` removes the metadata associated to a pointer from the metadata store. Once the metadata is deleted, any `pct_auth` to the deleted memory will fail even if the memory is re-allocated. This API should be called whenever memory (whether on the heap or the stack) is deallocated.

4.4.3 Pointer Operations

Since a PACTIGHT-signed pointer has a PAC in its upper bits, care must be taken to not break the semantics of existing C/C++ pointer semantics. In particular, we take care of the following four cases:

1) PACTIGHT-signed pointer comparison: Even if two pointers refer to the same memory address, their PACs are different since the locations of the two pointers are different (*i.e.*, `&p!=&q`). Hence, PACTIGHT strips off the PAC from the PACTIGHT-signed pointer before comparison by looking for the `icmp` instruction.

2) PACTIGHT-signed pointer assignment: When assigning one signed pointer (source) to another signed pointer (target), the target pointer should be signed again with its location.

3) PACTIGHT-signed pointer argument: There are functions that directly manipulate a pointer. For example, `munmap` and `free` take a pointer as an argument and deallocate a virtual address segment or a memory block for a given address. If their implementations do not consider PAC-signed pointers, passing a PAC/PACTIGHT-signed pointer can cause segmentation fault. For those functions, PACTIGHT strips off the PAC before passing the signed pointer as an argument.

4) PACTIGHT-signed pointer arithmetic: PACTIGHT supports pointer arithmetic on arrays. PACTIGHT assigns the same random tag for all elements in an array, with the metadata keeping track of the size of an element and the number of elements in the array to efficiently enforce spatial safety. Figure 5

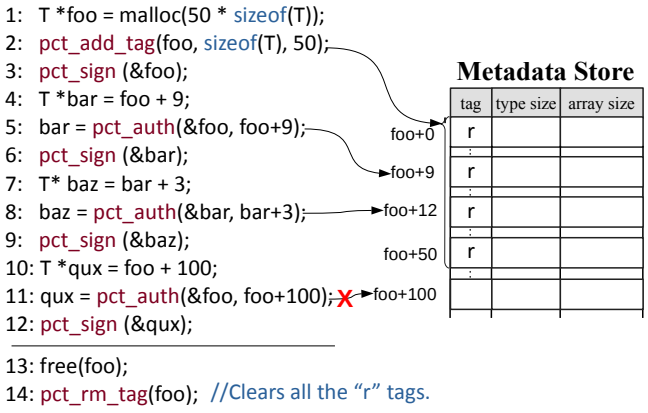

Figure 5: Example of PACTIGHT handling array operations.

shows a simplified representation of the metadata. PACTIGHT first assigns the same random tag `r` to all 50 array elements after the array allocation (Line 2). Each element has its own metadata. In Line 5, PACTIGHT successfully authenticates `foo+9` using `pct_auth(&foo,foo+9)`. PACTIGHT successfully authenticates `bar+3` in Line 8, since `bar+3` is `foo+12`, and is within the array boundary. On the other hand, Line 10 violates spatial memory safety, and PACTIGHT throws an exception at Line 11. This is because `tag(foo+100)` either does not exist or has a different tag. Figure 11 shows the code of the runtime library.

The mechanism works the same for temporal memory safety; a freed object will not have a tag (Line 14), and newly allocated objects in the same location will have a different tag. Thereby, PACTIGHT can effectively reject spatial and temporal memory violations. Note that PTAuth [27] performs "backward search" to find an array base address, which is not necessary in PACTIGHT.

4.4.4 Metadata Store

PACTIGHT maintains a metadata store for allocated memory objects. For each allocated memory object, the metadata store maintains a random tag, the size of each individual element (or type size), and the number of elements in an array (or array size). Non-array objects will be treated as an array having a single element. We use either a 64-bit (default) or a 32-bit tag and we compare the memory overhead between both tag sizes in §7.3.1.

We implemented the metadata store as a linear open addressing hash table (base + offset) using the address (*i.e.*, `p`) as the key. The base address is kept in a reserved register, `X18`, to avoid leaking the metadata location (*i.e.*, stack spill). The metadata store is initialized when the program starts and is maintained by PACTIGHT's runtime library. An entry in the metadata store is allocated and deallocated using `pct_add_tag` and `pct_rm_tag`, respectively. Whenever PACTIGHT needs to sign or authenticate (`pct_sign`, `pct_auth`), it looks up the metadata store to get the associated random tag and to check if the accessed memory is valid or not. PACTIGHT relies on sparse address space support of the OS.

4.4.5 A Running Example

The code snippet in Figure 2 (right) shows how PACTIGHT APIs are used to protect a local function pointer `p`. When an object gets allocated (Line 2), `pct_add_tag` allocates the metadata by setting a random tag and all the associated metadata. The number of elements and type size can be determined statically by analyzing the LLVM IR. Whenever a stack variable is assigned, the PAC is added with `pct_sign` (Lines 7, 8). If the pointer is dereferenced (Line 13) or if any change in assignment happens to the pointer legitimately, the PAC is authenticated (`pct_auth`) (Line 12) and a new PAC is generated for the new pointer with `pct_sign` (Line 14). When a pointer gets deallocated (after the `return` on Line 16, since we are on the stack), the pointer is authenticated and all metadata is removed (`pct_rm_tag`). This is done by reading the type size and array size from the metadata and removing the metadata accordingly.

5 PACTIGHT Defense Mechanisms

This section presents the PACTIGHT defense mechanisms built on top of the PACTIGHT runtime. The PACTIGHT compiler passes automatically instrument all globals, stack variables, and heap variables, inserting the necessary PACTIGHT APIs. We implement four defense mechanisms: 1) Control-Flow Integrity (forward edge protection), 2) C++ VTable protection, 3) Code Pointer Integrity (all sensitive pointer protection), and 4) return address protection (backward edge protection).

5.1 Control Flow Integrity (PACTIGHT-CFI)

PACTIGHT-CFI guarantees forward-edge control-flow integrity by ensuring the PACTIGHT pointer integrity properties for all code pointers. It authenticates the PAC on a function pointer at legitimate function call sites. At all other sites, the code pointer is sealed with the PACTIGHT signing mechanism so it cannot be abused. Any direct use of a PACTIGHT-signed pointer results in a segmentation fault, causing illegal memory access.

Instrumentation overview. In order to prevent any misuse and enforce all three security properties for a code pointer, PACTIGHT-CFI should set metadata upon allocation and remove it upon deallocation. Also, a function pointer should always be authenticated before every le-

gitimate use and it should be signed again afterwards. The PACTIGHT-CFI instrumentation passes accurately identify and instrument all instructions in the LLVM IR that allocate, write, use, and deallocate code pointers.

Identifying code pointers. PACTIGHT-CFI identifies all code pointers using LLVM type information. Since code pointers can be present inside composite types (*e.g.*, `struct` or an array of `struct`), PACTIGHT-CFI also recursively looks through all elements inside a composite type. We specially handle the case that a code pointer is manipulated after it is converted to some universal pointer type (*e.g.*, `void*`). For example, for `memcpy` and `munmap` which take `void*` arguments, PACTIGHT-CFI gets the actual operand type first and instrumentation is done accordingly. This is not only done for `memcpy` and `munmap`, but for all universal pointer types. We look ahead for when they are typecasted (*i.e.*, `BitCast` in LLVM IR) to get the original type accordingly

Instrumenting PACTIGHT APIs. Setting the metadata by instrumenting `pct_add_tag` is done immediately after all code pointer allocations. This is done for all global, stack and heap variables. In the case of initialized global variables, `pct_add_tag` and `pct_sign` are appended to the global constructors. In this way, PACTIGHT-CFI maintains the appropriate metadata for all global variables during program execution.

If the destination operand of the `store` instruction is a code pointer, `pct_sign` is instrumented right after the `store` instruction to sign the code pointer.

`pct_auth` must be called before any use of a code pointer. Specifically, PACTIGHT-CFI looks for the relevant `load` and `call` instructions and it instruments `pct_auth` immediately before the instructions. If the authentication fails, the top two bits of the pointer are flipped meaning any use of the pointer causes a segmentation fault, effectively denying any attack. As the PAC authentication instructions (*e.g.*, `autia`) strips off the PAC, the PAC should be added again after the function call. Thus, PACTIGHT-CFI replaces the stripped pointer with the signed version after indirect `call` instructions. This ensures that a PAC is always present.

Whenever a code pointer is deallocated (*e.g.*, `free`, `munmap`), PACTIGHT-CFI removes the metadata by instrumenting `pct_rm_tag` before the deallocation. For stack variables, `pct_rm_tag` is instrumented right before `return`, and it removes the metadata from the entire stack frame at once, from the first variable to the last variable that has any metadata set.

Summary. PACTIGHT-CFI is precise and efficient by enforcing the PACTIGHT pointer integrity properties and leveraging hardware-based PA. Moreover, it provides the Unique Code Target (UCT) property [33] because ensuring the PACTIGHT pointer integrity properties implies that the equivalence class (EC) size (*i.e.*, the number of allowed legitimate targets at one call site) is always one. Thus, it defends against all ConFIRM [40] attacks, which essentially rely on the presence of more than one legitimate targets in an EC and replace an indirect call/jump target with another allowed target.

5.2 C++ VTable Protection (PACTIGHT-VTable)

C++ relies on virtual functions to achieve dynamic polymorphism. At every virtual function call, a proper function is used in accordance with the object type. The mapping of an object type to a virtual function is done by the use of a virtual function table (VTable) pointer, which is a pointer to an array of virtual function pointers per object type. A VTable pointer is initialized in the object's constructor and it is valid until an object is destructed. Attacking the virtual function table pointer is a common exploit in C++ programs [18, 54, 63].

Identifying VTable pointers. PACTIGHT-VTable identifies a VTable pointer in a C++ object by analyzing types in LLVM. It investigates all composite types and checks if it is a class type having one or more virtual functions. If so, it marks the first hidden member of the class as a VTable pointer. PACTIGHT-VTable also handles `dynamic_cast<T>`, since `dynamic_cast<T>` is only valid for a class with at least one virtual function pointer, so it has a virtual function table, and thereby they all are already considered sensitive types.

Instrumenting PACTIGHT APIs. Upon a C++ type having a virtual function allocated, PACTIGHT-VTable instruments `pct_add_tag`.It instruments `pct_sign` immediately after the VTable pointer is assigned by the object's constructor. This adds the PAC to the pointer to seal it. Then, `pct_auth` is instrumented right before `load`ing the VTable pointer. A failed authentication flips the top two bits of the pointer, rendering it unusable. Correspondingly, `pct_rm_tag` is instrumented right before the object is destroyed (deallocation).

5.3 Code Pointer Integrity (PACTIGHT-CPI)

PACTIGHT-CPI increases the coverage of PACTIGHT-CFI to guarantee integrity of all sensitive pointers [38]. Sensitive pointers are all code pointers (*i.e.*, PACTIGHT-CFI coverage) and all data pointers that point to code pointers. It is possible to hijack

```
1  /** ==== nginx/http/ngx_http_variables.h ==== */
2  typedef struct ngx_http_variable_s ngx_http_variable_t;
3
4  // a function pointer type (i.e., sensitive type)
5  typedef ngx_int_t (*ngx_http_get_variable_pt)(...);
6
7  struct ngx_http_variable_s {
8      ngx_str_t                name;
9      // sensitive function pointer
10     ngx_http_get_variable_pt  get_handler;
11     // ...
12 }; // a sensitive data type
```

Figure 6: Example of a sensitive data pointer in the (simplified) NGINX source code.

control-flow by corrupting a sensitive *data* pointer because it can reach a code pointer. Figure 6 shows an example of sensitive pointers from NGINX. A function pointer type `ngx_http_get_variable_pt` at Line 5 is a sensitive code pointer. Also, a `struct` type `ngx_http_variable_s` at Line 7 is a sensitive data type because it has another sensitive pointer (`get_handler` at Line 10) in it. If a sensitive data pointer or its array index are corrupted, an attacker can hijack the control-flow without directly corrupting the function pointer.

Identifying sensitive pointers. PACTIGHT-CPI expands the type analysis of PACTIGHT-CFI to include all sensitive pointers. It classifies a composite type that contains a function pointer as a sensitive type. Then, it recursively classifies a composite type that contains any sensitive pointer in it as a sensitive type until it cannot find any more sensitive types. We over-approximate when detecting security-sensitive pointers. That is, we regard a pointer as security-sensitive if we cannot determine if it is non-security-sensitive statically (*e.g.*, C union). This approach may add extra instrumentation, however, it will not compromise PACTIGHT 's security guarantees.

Instrumenting PACTIGHT APIs. Instrumentation is then done in a similar manner to PACTIGHT-CFI by instrumenting all instructions that allocate, store, modify and use sensitive pointers. In case the pointers are of universal type (*i.e.*, `void*` or `char*`), PACTIGHT-CPI gets its actual type by looking ahead for a typecast and then instrumentation is done accordingly.

5.4 Return Address Protection (PACTIGHT-RET)

Protecting return addresses is critical because they are, after all, the root of ROP attacks. Meanwhile, the return address protection scheme should impose minimal performance overhead because function call/return is very frequent during program execution. We aim to minimize the signing/authentication overhead without compromising the PACTIGHT pointer integrity properties.

No non-dangling in return address. One interesting fact is that a return address cannot be a dangling pointer.[1] Hence, the non-dangling property doesn't need to be enforced and random tags are unnecessary. Not using a tag offers large performance benefits as metadata store lookup cost to get the random tag can be removed.

Binding all previous return addresses. Instead of blending the location of a return address in a stack to provide the *non-copyability* property, we use the *signed* return address of a previous stack frame. Since the stack distance to a return address in a previous stack frame is determined at compile time, accessing the previous return address with a constant offset binds the current return address to the relative offset of the previous stack frame (*i.e.*, the current stack frame). Hence we can achieve the *non-copyability* property for return addresses. In addition, by blending the signed return address of a previous stack frame, we chain all previous return addresses to calculate the PAC of the current return address. This approach is inspired by PACStack [42]. Both PACTIGHT-RET and PACStack incorporate the entire callstack in the modifier to prevent the reuse attack. In regards to dynamic stack allocation, the `alloca()` function can dynamically adjust the stack frame size. To support dynamic stack allocation, PACTIGHT-RET uses LLVM intrinsics, such as `getFrameInfo()` and `getCalleeSavedInfo()`, that allow us to find the previous stack frame and calculate the distance correctly.

Signing and authentication of a return address. Our optimized sign/authentication scheme for return addresses is as follows. We blend a caller's unique function ID and the signed return address from the previous stack frame to generate the modifier. This blending allows us to achieve the *non-copyability* property by chaining all previous return addresses (binding a return address to a control-flow path), alongside the guarantee of the *unforgeability* property achieved by the PAC mechanism. Instrumentation is done in the MachineIR level during frame lowering. Frame lowering emits the function prologues and epilogues. The PAC is added at the function prologue and authenticated at the function epilogue. The LLVM-assigned function ID is unique due to the use of link time optimization (LTO).

5.5 Optimization to Reduce PAC Instructions

The main source of overhead in PACTIGHT would be due to the cryptographic operations done by the

[1] Precisely speaking, a return address can be a dangling pointer for Just-In-Time (JIT) compiled code in a managed runtime (*e.g.*, Java, Python). However, protecting control-flow hijacking in a managed runtime is the out of scope for PACTIGHT.

QARMA algorithm. This is done every time a PAC instruction is executed. As discussed in §4.4.5, `pct_auth` strips the PAC from the pointer and `pct_sign` is added again after the pointer is used to add the PAC again, thus maintaining the seal on the pointer. Thus, instead of re-adding the PAC with `pct_sign`, we save the original pointer with the PAC before `pct_auth` in a temporary register, and overwrite the stripped pointer with a PACed pointer without needing to call `pct_sign`. Note that our code generation pass prevents the stack spill of the temporary register to avoid the register from being restored.

6 Implementation

Our prototype consists of 4014 lines of code (LoC), with 3237 LoC for the LLVM pass, 656 LoC for the PACTIGHT runtime library, and 121 LoC for the AArch64 backend. PACTIGHT-CFI, PACTIGHT-VTable and PACTIGHT-CPI are all implemented in the LLVM IR level while PACTIGHT-RET is implemented in the AArch64 backend. The PACTIGHT runtime library is integrated with LLVM as part of `compiler_rt`. We use CSPRNG seeded by hardware RNG (RNDR in ARMv8) [4] for random tag generation. To harden our prototype, we used different key types for sensitive function pointers (`pacia`, `autia`), sensitive data pointers (`pacda`, `autda`), and return addresses (`pacib`, `autib`).

We apply several optimizations to PACTIGHT. First, we use Link Time Optimization (LTO), which combines all the object files into one file. Then, we inline all our PACTIGHT runtime library functions. Finally, we implement the additional optimization, discussed in §5.5, to reduce PAC instructions. The evaluation of the impact of these optimizations is discussed in §7.3.2. We also make our code for PACTIGHT public at `https://github.com/cosmoss-jigu/pactight`.

7 Evaluation

We evaluate PACTIGHT by answering the following:

- How effectively can PACTIGHT prevent not only synthetic attacks but also real-world attacks by enforcing PACTIGHT pointer integrity properties? (§7.2)
- How much performance and memory overhead does PACTIGHT impose? (§7.3)

7.1 Evaluation Methodology

Evaluation environment. We ran all evaluations on Apple's M1 processor [10], which is the only commercially available processor supporting ARMv8.4 architecture with ARM PA instructions. Specifically, we used an Apple Mac Mini M1 [9] equipped with 8GB DRAM, 4 big cores, and 4 small cores. We ported our prototype to Apple's LLVM 10 fork [1]. For all applications, we enabled `O2` and `LTO` optimizations for fair comparison.

Evaluation of C applications. We ran all C applications with real ARM PA instructions. In this case, we turned off all Apple LLVM's use of PA [11] to avoid the conflicting use of PA instructions.

Evaluation of C++ applications. During initial evaluation, we found that the use of PA instructions is built into Apple's standard C++ library. We have investigated using Ubuntu Linux [21] on the M1 to work around this problem. At time of writing, the Linux kernel on Ubuntu/M1 does not support PA – the kernel does not activate PA during the boot procedure – so userspace applications cannot use PA instructions.

For C++ applications, we use two different approaches to validate if PACTIGHT's instrumentation is correct and to get an accurate performance estimation. For the correctness testing, we ran all C++ applications on ARM Fixed Virtual Platform (FVP) [12], which is an ARM hardware platform simulator that supports pointer authentication. We used the FVP only to test correctness, since it is not a cycle-accurate simulator. We ran Linux on FVP to run C++ applications, and we modified the Linux kernel and bootloader to activate ARM PA. All our C++ applications passed the correctness testing with FVP. To simulate the overhead of a PA instruction and to get accurate performance estimates on real hardware, we measured the time to execute a PA instruction and found that seven XOR (`eor`) instructions take almost the same time – 0.15% faster – to execute one PA instruction on the Apple Mac Mini M1. Similarly, Lilijestrand *et al.* [43] also replaced a PA instruction with four `eor` instructions to estimate the performance overhead, which is more optimistic than our measurement on hardware.

7.2 Security Evaluation

In this section, we evaluate PACTIGHT's effectiveness in stopping security attacks using three real-world exploits (§7.2.1) and five synthesized exploits (§7.2.2).

7.2.1 Real-World Exploits

We evaluated PACTIGHT with three real-world exploits to test its effectiveness against real vulnerabilities.

(1) CVE 2015-8668. This is a heap-based buffer overflow [24] corrupting a sensitive pointer in the `libtiff` library. The heap overflow overwrites a function pointer in the `TIFF` structure, which allows attackers to achieve arbitrary code execution. PACTIGHT-CFI/CPI success-

fully detects this and stops it from completing by enforcing `pct_auth` on the corrupted function pointer.

(2) CVE-2019-7317. This is a use-after-free exploit [25] in the `libpng` [3] library. The `png_image_free` function is called indirectly and frees memory that is referenced by `image`, a sensitive pointer. `image` is then dereferenced. Since PACTIGHT-CPI does recursive identification, `image` is instrumented. When `image` gets dereferenced after the free, PACTIGHT-CPI will detect that no metadata exists and halts the execution.

(3) CVE-2014-1912. This is a buffer overflow vulnerability [55] in `python2.7` that happens due to a missing buffer size check. An attacker can corrupt a function pointer in the `PyTypeObject` and achieve arbitrary code execution. PACTIGHT-CFI/CPI detects this by detecting the corrupted function pointer with `pct_auth`.

7.2.2 Synthesized Exploits

CFIXX test suite. We evaluated PACTIGHT with five synthesized attacks for C++ to demonstrate how PACTIGHT-VTable can defend against virtual function pointer hijacking attacks, COOP attacks [54] – an attack that crafts fake C++ objects. We used CFIXX C++ test suite [51] by Burow *et al.* [18]. It contains four virtual function pointer hijacking exploits (`FakeVT-sig`, `VTxchg-hier`, `FakeVT`, `VTxchg`) and one COOP exploit. To make the test suite more similar to real attacks, we modified the suite to use a heap-based overflow rather than directly overwriting with `memcpy`. This modification is similar to a synthesized exploit in OS-CFI [36]. PACTIGHT-VTable detects all the exploits by enforcing `pct_auth` on the virtual function pointer before the virtual function call. The COOP attack crafts a fake object without calling the constructor and utilizes a virtual function pointer of the fake object. PACTIGHT-VTable detects this due to the fact that it was never initialized and thus `pct_auth` fails.

Vulnerable code to other PAC defenses. We describe here a synthesized exploit that bypasses PARTS [43] and PTAuth [27], relying on the security guarantees provided by the *non-copyability* property. The security benefits of the non-copyability property are demonstrated by the PAC reuse attack in the vulnerable code in Figure 7. PARTS [43] is vulnerable to this attack while PACTIGHT is not. If two pointers have the same modifier (type-id in PARTS) and point to the same address, then the processor will generate the same PAC, and thus they can be used interchangeably at a different code location. This is possible in PARTS if both pointers have the same LLVM *ElementType*. This is similar in concept to the COOP attack in terms of pointer manipulation. Our incorporation of a pointer location (`&p`) into the modifier with the non-copyability property blocks this attack by binding a signed PAC to a specific pointer location in the code. This binding will not allow a signed pointer to be used from a different pointer location.

```
1  T foo,bar;
2  foo.funcptr = &printf;
3  bar.funcptr = &system;
4  T *p = &foo; // p stores a valid PAC of foo
5  T *q = &bar; // q stores a valid PAC of bar
6  // An attacker performs arbitrary read/write here
7  // (by exploiting a known vulnerability)
8  // to overwrite p as q, i.e., p = q;
9  // now p stores a valid PAC of &bar
10 p->funcptr();  // Runs system() in PARTS
11               // because type of p and q are the same
```

Figure 7: Example of vulnerable code that PACTIGHT defends against but PARTS [43] cannot.

7.3 Performance Evaluation

7.3.1 Benchmarks

Benchmark applications. For our performance evaluation, we use three benchmarks: SPEC CPU2006 [32], nbench [47], and CoreMark [2], and one real-world application, NGINX web server [5]. In order to run the SPEC CPU2006 benchmark suite, we ported each SPEC benchmark to Apple M1 and built it from scratch. We were not able to run one benchmark, `403.gcc`, on the Apple M1 even with Apple's vanilla Clang/LLVM compiler. We suspect a bug in the MacOS/M1 toolchain. We ran all benchmark applications with real PA instructions except for seven C++ benchmarks in the SPEC benchmark. For the C++ benchmarks, we replaced a PA instruction with seven `eor` instructions to emulate the overhead of the PA instructions as discussed in §7.1.

Performance overhead. Figure 8 shows the performance of the PACTIGHT defenses on the individual SPEC benchmarks, nbench, and CoreMark. The SPEC benchmarks have a geometric mean of 0.64%, 1.57%, and 5.77% for PACTIGHT-RET, PACTIGHT-CFI+VTable+RET, and PACTIGHT-CPI, respectively. The geometric means of all benchmark applications are 0.43%, 1.09%, and 4.07% for PACTIGHT-RET, PACTIGHT-CFI+VTable+RET, and PACTIGHT-CPI, respectively. As can be seen, PACTIGHT has very low overhead on almost all benchmarks and across all the protection mechanisms. The exceptions here are `453.povray`, `471.omnetpp`, and `483.xalancbmk` for PACTIGHT-CPI. We discuss these further in §A.4.

We evaluated NGINX on the Apple M1 using its 4 big cores to stress the machine. We used the same configu-

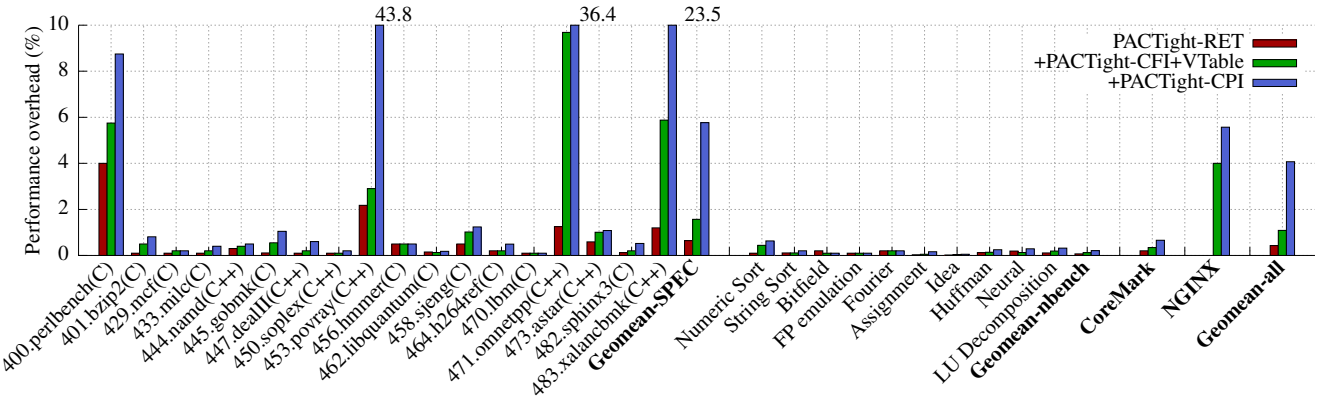

Figure 8: The performance overhead of SPEC CPU2006, nbench, and CoreMark relative to an unprotected baseline build. Our three PACTIGHT protections are: 1) return addresses (PACTIGHT-RET), 2) CFI, C++ VTable protection and return addresses (+PACTIGHT-CFI+VTable), and 3) CPI offering full protection for all sensitive pointers (+PACTIGHT-CPI).

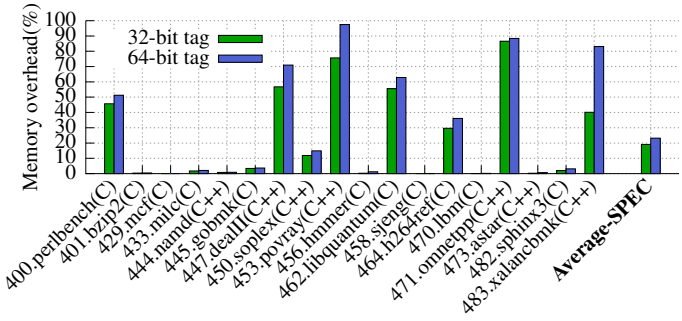

Figure 9: The memory overhead of SPEC CPU2006 for PACTIGHT-CPI, PACTIGHT's highest protection mechanism. PACTIGHT imposes a low overhead of 23.2% and 19.1% on average for 64-bit and 32-bit tags, respectively.

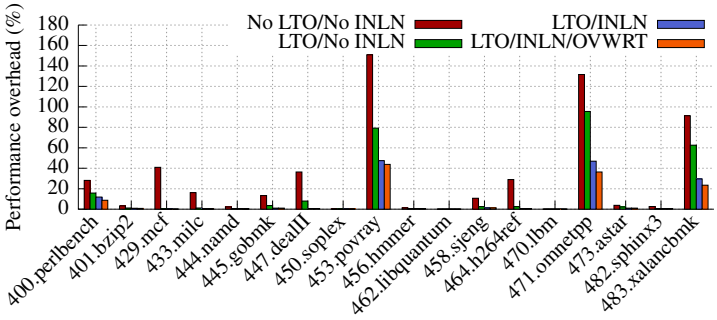

Figure 10: Impact of the performance optimizations. (LTO: Link Time Optimization INLN: inlining APIs; OVWRT: overwrite stripped pointer with PACed pointer)

ration used to bench NGINX TLS transactions per second [48]. We used the HTTP benchmarking tool `wrk` [29] to generate concurrent HTTP requests. We ran `wrk` on another machine under the same network. Each `wrk` spawns three threads where each thread handles 50 connections. We observed a small performance overhead: 4% for PACTIGHT-CFI and 5.57% for PACTIGHT-CPI.

Memory overhead. In order to see how much additional memory is used by PACTIGHT's metadata store, we measured the maximum resident set size (RSS) during the execution of the SPEC CPU2006 benchmarks. We ran the SPEC benchmarks with the PACTIGHT-CPI protection because it is the highest level of protection in PACTIGHT, thus it requires the largest number of entries in the metadata store. We used both 64-bit and 32-bit tag sizes to measure the gain if we used a smaller tag. The size of the metadata is 16 bytes in the case of a 64-bit tag, and 12 bytes in the case of a 32-bit tag. Figure 9 shows the results of our measurements. In spite of measuring the highest security mechanism with the most instrumentations, PACTIGHT imposes an overhead of 23% on average for 64-bit tags and 19% on average for 32-bit tags. The memory overhead is proportional to $O(n)$ where n is the number of sensitive pointers, with the metadata size being either $2\times$ the size of the pointer (64-bit tag) or $1.5\times$ the size of the pointer (32-bit tag).

7.3.2 Impact of Optimizations

Here we showcase the impact of the optimizations discussed in §5.5 and §6. We added three optimizations to PACTIGHT to improve performance: Link Time Optimization (LTO), inlining of PACTIGHT runtime library functions (INLN), and overwriting the stripped pointer with a PACed pointer (OVWRT). Figure 10 shows the performance overhead of PACTIGHT-CPI with and without the optimizations in various configurations. As can be seen, the optimizations were critical to greatly improving PACTIGHT's performance.

8 Discussion and Limitations

Information leakage attack on the metadata store. In our threat model, an attacker is able to access the PACTIGHT metadata store while it is probabilistically hidden using address space layout randomization (ASLR). Even if the PACTIGHT metadata is leaked, an

attacker is not able to exploit the leaked information. In order for an attacker to take advantage of the leakage, she has to launch an attack from a different location and this is already protected by the non-copyability property. The only part of the modifier that gets leaked is the random tag, but the location (`&p`) in the modifier still enforces the non-copyability property. In regards to legal and illegal pointers, PACTIGHT always authenticates the right hand side of a PACTight-signed pointer assignment. Thus, if a pointer in the right hand side is illegal, its authentication will fail. In this way, PACTIGHT prevents the propagation of illegal pointers. Another hypothetical case is that an attacker reuses a random tag to bypass the non-dangling property. While such an attack is possible in theory, the bar is very high in practice. An attacker cannot reuse dangling pointers at an arbitrary location due to the non-copyability property, and this significantly limits the attack. Moreover, we argue this is not a fundamental flaw in PACTIGHT's design. The random tag can be enforced using ARM's new Memory Tagging Extension (MTE) feature [13]. The presence of MTE will mitigate the random tag reuse attacks since the tags are protected in physical memory that can never be accessed by an attacker. PACTIGHT can easily be extended to utilize MTE as a tag store.

9 Related Work

In this section, we only discuss related studies that have not been discussed previously .

Cryptographic pointer defenses. CCFI [46] uses MACs to protect return addresses, function pointers, and VTable pointers. Conceptually, the use of MACs is similar to PA. But, since CCFI does not benefit from the hardware-accelerated PA instructions, it has an average of 52% overhead across SPEC CPU2006 benchmarks.

Integrity policies. Control-Flow Integrity (CFI) [6] restricts the valid target sites for indirect control-flow transfers. Static CFI schemes are vulnerable to control-flow bending [19]. Since PACTIGHT-CFI-VTable seals a pointer with its location and a random tag, this limits the feasibility of a reuse attack. Other dynamic approaches require additional threads to analyze data from Intel Processor Trace [28, 31, 33, 45] limiting scalability.

Code Pointer Integrity (CPI) [38] protects sensitive pointers (code pointers and pointers that refer to code pointers) by storing the sensitive pointers in a seperate hidden memory region. Return addresses are stored on a safe stack. PACTIGHT-CPI provides temporal safety to sensitive pointers, which CPI does not, and protects virtual function pointers in addition to sensitive pointers, all while having a lower overhead across all defenses.

CFIXX [18] protects VTable pointers by enforcing Object Type Integrity (OTI). CFIXX stores metadata on construction and checks the metadata at the virtual function call site. CFIXX incurs an overhead of 4.98%. PACTIGHT-CFI+VTable incurs lower overhead (1.98%) whilst providing stronger guarantees by enforcing CFI.

Temporal memory safety. Explicit pointer invalidation is a common strategy to enforce temporal memory safety. DangNull [39], DangSan [60], FreeSentry [62], pSweeper [44], and BOGO [64] invalidate all pointers to an object when the object is freed. These schemes typically incur high costs. CRCount [57] implicitly invalidates pointers by using reference counting. This approach comes at memory costs since some objects may never be freed. CETS [50] uses disjoint metadata to check if an object still exists upon pointer dereferences. MarkUs [7] is a memory allocator that protects from use-after-free attacks. It quarantines freed data and prevents reallocation until there are no dangling pointers. In contrast, PACTIGHT offers broader protection and protects sensitive pointers from memory corruption attacks.

10 Conclusion

We presented PACTIGHT, an efficient and robust mechanism to guarantee pointer integrity using ARM's Pointer Authentication mechanism. We identified three security properties PACTIGHT enforces to ensure pointer integrity: (1) Unforgeability: a pointer cannot be forged to point to an unintended memory object. (2) Non-copyability: a pointer cannot be copied and re-used maliciously. (3) Non-dangling: a pointer cannot refer to an unintended memory object if the object has been freed. We implememented PACTIGHT with four defense mechanisms, protecting forward edge, backward edge, virtual function pointers, and sensitive pointers. We demonstrated the security of PACTIGHT against real and synthesized attacks and showcased its low performance and memory overhead, 4.07% and 23.2%, on average respectively, using real PAC instructions.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and feedback. This work is supported in part by the U.S. Office of Naval Research under grants N00014-18-1-2022, the U.S. Nuclear Regulatory Commission under grants 31310021M0005, and the Federal Aviation Administration under grants A51-A11L.UAS.92.

References

[1] Apple unleashes M1. https://github.com/apple/swift-llvm.

[2] CoreMark - An EEMBC Benchmark. https://www.eembc.org/coremark.

[3] libpng. http://www.libpng.org/pub/png/libpng.html.

[4] RNDR, Random Number. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/RNDR--Random-Number.

[5] NGINX Web Server, 2019. nginx.org/.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[7] Sam Ainsworth and Timothy M. Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2020.

[8] Amazon. Optimized cost and performance for scale-out workloads, 2021. https://aws.amazon.com/ec2/instance-types/a1/.

[9] Apple. Apple Mac Mini M1, 2020. https://www.apple.com/shop/buy-mac/mac-mini/apple-m1-chip-with-8-core-cpu-and-8-core-gpu-256gb.

[10] Apple. Apple unleashes M1, 2020. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[11] Apple. Operating system integrity, 2021. https://support.apple.com/en-hk/guide/security/sec8b776536b/1/web.

[12] Arm. Fixed Virtual Platforms. https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms.

[13] Arm. Memory Tagging Extension, 2019. https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[14] Roberto Avanzi. The QARMA Block Cipher Family, 2016. https://eprint.iacr.org/2016/444.pdf.

[15] Brandon Azad. Examining Pointer Authentication on the iPhone XS, 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[16] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

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

[18] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[20] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] Corellium. How We Ported Linux to the M1, 2021. https://corellium.com/blog/linux-m1.

[22] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[23] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea Perez, and Jan-Erik Ekberg. Camouflage: Hardware-assisted CFI for the ARM linux kernel. In *Proceedings of the 57th Annual Design Automation Conference (DAC)*, June 2020.

[24] Dongliang Mu. CVE-2015-8668, 2018. cve-2015-8668-exploit.

[25] Eddie Lee. CVE-2019-7317, 2019. cve-2019-7317-exploit.

[26] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, page 901–913, Denver, Colorado, October 2015.

[27] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *Proceedings of the 30th USENIX Security Symposium (Security)*, August 2021.

[28] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

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

[30] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[31] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[32] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. URL: http://doi.acm.org/10.1145/1186736.1186737, doi:10.1145/1186736.1186737.

[33] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

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

[35] Jonathan Corbet. x86 NX support, 2004. https://lwn.net/Articles/87814/.

[36] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[38] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[39] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[40] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[41] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with paced canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution (SysTEX)*, pages 4:1 – 4:6, Huntsville, Ontario, October 2019.

[42] Hans Liljestrand, Lachlan J. Gunn, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack. In *Proceedings of the 30th USENIX Security Symposium (Security)*, August 2021.

[43] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 177–194, Santa Clara, CA, August 2019.

[44] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 1635–1648, Toronto, ON, Canada, October 2018.

[45] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[46] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[47] Uwe Mayer. Linux/Unix nbench, 2017. https://www.math.utah.edu/~mayer/linux/bmark.html.

[48] Faisal Memon. NGINX Plus Sizing Guide: How We Tested, 2016. https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/.

[49] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[50] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

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

[52] Oracle. Advancing the Future of Cloud with Arm-Based Computing, 2021. https://www.oracle.com/events/live/advancing-future-cloud-arm-based-computing/.

[53] Qualcomm. Qualcomm's 48-Core ARMv8 Processor Runs Windows Server, 2017. https://www.electronicdesign.com/technologies/embedded-revolution/article/21805493/qualcomms-48core-armv8-processor-runs-windows-server.

[54] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[55] @sha0coder. Python - 'socket.recvfrom_into()' Remote Buffer Overflow, 2014. URL: https://www.exploit-db.com/exploits/31875.

[56] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[57] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[58] Nigel P. Smart. Cryptography Made Simple, 2015. Section 1.4.2.

[59] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[60] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 405–419, Belgrade, Serbia, April 2017.

[61] James Vincent. Apple calls A12 Bionic chip 'the smartest and most powerful chip ever in a smartphone', 2018. `https://www.theverge.com/circuitbreaker/2018/9/12/17826338/apple-iphone-a12-processor-chip-bionic-specs-speed`.

[62] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[63] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[64] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.

Appendix

We describe the PACTIGHT runtime library functions in detail (§A.1), its collision likelihood (§A.2), showcase PACTIGHT instrumentation statistics (§A.3), and finally analyze the high overhead benchmarks (§A.4).

A.1 PACTIGHT Runtime Library Functions

Figure 11 presents the (simplified) PACTIGHT library function code. The two omitted functions, `meta_hashtable_get` and `meta_hashtable_set`, retrieve and set the metadata, respectively.

```
/** == Pseudocode for the library
 * functions in PACTight== **/

pct_add_tag(void* p, int tsz, int asz){
   tag = CSPRNG(); // Generate random tag
   for(int i = 0; i < asz; i++){
      meta_hashtable_set(p, tag, tsz, asz - i);
      p = p + tsz;
   }
}

pct_sign(void** p){
   metadata* meta = meta_hashtable_get(*p);
   pac_modifier = p ^ meta->tag;
   __asm volatile ("pacia %x[pointer], %x[modifier]\n"
                   : [pointer] "+r" (*p)
                   : [modifier] "r"(pac_modifier)
                  );
}

pct_auth(void** p, void* p_index){
   metadata* meta = meta_hashtable_get(p_index);
   pac_modifier = p ^ meta->tag;
   __asm volatile ("autia %x[pointer], %x[modifier]\n"
                   : [pointer] "+r" (*p)
                   : [modifier] "r"(pac_modifier)
                   );
}

pct_rm_tag(void* p){
   metadata* meta = meta_hashtable_get(p);
   int asz = meta->asz;
   int tsz = meta->tsz;
   for(int i = 0; i < asz; i++){
      meta_hashtable_remove(p);
      p = p + tsz;
   }
}
```

Figure 11: PACTIGHT runtime library functions.

A.2 Collision Likelihood of PACTIGHT

A determined attacker can attempt to break our PACTIGHT scheme using modifier collisions. For example, if an attacker allocates a PAC'd pointer `p` with random tag `A` at location `B`, then deallocates and reallocates `p` with a different random tag `C`. Then, the attacker can reuse `p` at a different location `D` if (`A` XOR `B`) collides with (`C` XOR `D`). The probability that this collision occurs is extremely low. Because we XOR the modifier with a 64-bit random tag, the distribution of PAC modifiers is uniformly random with 64 bits of entropy (*i.e.*, 2^{64}); therefore, an attacker cannot practically break the non-copyability property via modifier collisions.

Alternatively, an attacker can attempt to break the scheme using PAC collisions. By reusing a sensitive pointer at many different locations, there is a chance (albeit a very low probability) that the same PAC could be generated even though the modifiers are different. This would require an expected 2^b guesses, where b is the number of bits available for PAC ($b = 16$ in ARMv8.3-A). The birthday problem [58] does not apply in this case since an attacker has no way to efficiently bruteforce many pointers at the same time. This attack is equivalent to attempting to forge an authenticated pointer. Consequently, these collision attacks are not feasible against the metadata scheme that PACTIGHT proposes.

A.3 Instrumentation Statistics

Table 2 shows various instrumentation statistics for PACTIGHT-CPI in SPEC CPU2006. These include compilation time, binary size, the total and protected number of loads and stores, and the number of instrumentations of `pct_add_tag`, `pct_sign`, `pct_auth` and `pct_rm_tag`. As shown, PACTIGHT-CPI imposes a marginal overhead in compilation time increase and binary size increase, 3.14% and 13.87%, respectively. For some benchmarks, the overhead is not directly proportional to the number of instrumentations. This is because the instrumentations may be called several times in a loop for example.

There are a few of these benchmarks that show zero instrumentation. We investigated this and found that the compiler optimizes out the sensitive `load` and `store` instructions. Running PACTIGHT without compiler optimizations produces the instrumentations accordingly. Thus, there are no false negatives with these benchmarks.

A.4 Analysis on High Overhead Benchmarks

Three benchmarks in SPEC CPU2006, namely `453.povray`, `471.omnetpp` and `483.xalancbmk`, have high performance overhead with PACTIGHT-CPI than the rest of the benchmarks. In this section, we analyze why these benchmarks have higher overhead and suggest possible improvements.

`453.povray`. The overhead is mainly due to the loops using sensitive data pointers inside. Specifically, the `struct Method_struct` is one of the struct types in

Benchmark Name	Compilation time			Binary size			Number of stores	Number of protected stores	Percentage of protected stores	Number of loads	Number of protected loads	Percentage of protected loads	Number of pct_add_tag	Number of pct_sign	Number of pct_auth	Number of pct_rm_tag
	Vanilla	PACTight	Overhead	Vanilla	PACTight	Overhead										
400.perlbench (c)	318	366	15.09%	2616	2784	6.42%	15826	1083	6.84%	42753	9068	21.21%	310	1079	9064	46
401.bzip2 (c)	47	47	0.00%	208	240	15.38%	1754	11	0.63%	2602	150	5.76%	4	45	150	2
429.mcf (c)	25	25	0.00%	104	104	0.00%	252	0	0.00%	399	0	0.00%	0	0	0	0
433.milc (c)	120	122	1.67%	288	288	0.00%	889	16	1.80%	3201	35	1.09%	2	7	35	2
444.namd (c++)	114	115	0.88%	424	504	18.87%	2333	53	2.27%	6170	21	0.34%	41	64	21	35
445.gobmk (c)	286	297	3.70%	7696	8600	10.51%	4584	6	0.13%	17370	74	0.43%	8	10	74	6
447.dealII (c++)	1508	1516	0.53%	1488	1768	18.82%	41257	6204	15.04%	94791	8543	9.01%	2892	6745	8036	2867
450.soplex (c++)	408	434	6.37%	840	1072	27.62%	5409	254	4.70%	16665	724	4.34%	242	632	441	52
453.povray (c++)	625	653	4.48%	2496	3120	25.00%	15128	474	3.13%	25766	2247	8.72%	117	525	2029	36
456.hmmer (c)	141	148	4.96%	384	456	18.75%	3618	33	0.91%	8557	264	3.09%	16	28	264	16
462.libquantum (c)	32	33	3.13%	144	144	0.00%	270	0	0.00%	585	0	0.00%	0	0	0	0
458.sjeng (c)	61	62	1.61%	368	368	0.00%	1899	0	0.00%	3570	1	0.03%	1	1	1	1
464.h264ref (c)	296	304	2.70%	1248	1568	25.64%	11309	88	0.78%	27103	1659	6.12%	48	139	1663	11
470.lbm (c)	12	13	8.33%	104	104	0.00%	99	0	0.00%	269	0	0.00%	0	0	0	0
471.omnetpp (c++)	567	570	0.53%	2136	2528	18.35%	6007	1158	19.28%	8697	2890	33.23%	264	1206	2025	66
473.astar (c++)	34	35	2.86%	144	144	0.00%	708	0	0.00%	1191	2	0.17%	0	0	1	0
482.sphinx3 (c)	109	110	0.92%	384	416	8.33%	1421	20	1.41%	4716	152	3.22%	2	18	152	2
483.xalancbmk (c++)	3149	3624	15.08%	11184	19800	77.04%	39741	10834	27.26%	110595	35025	31.67%	3820	13207	33046	1065
Average/Total			**3.14%**			**13.87%**	**152504**	**20234**	**13.27%**	**375000**	**60855**	**16.23%**	**7767**	**23706**	**57002**	**4207**

Table 2: Instrumentation statistics for PACTIGHT-CPI in SPEC CPU2006.

`453.povray` that is considered to be sensitive. This struct has a series of function pointers and is used like a virtual function table. Pointers of type `struct Method_struct` and its members are used in loop conditions and inside loops. Since these pointers are sensitive, PACTIGHT-CPI enforces protection on them. This means calling `pct_auth` when they are dereferenced and re-adding the PAC with `pct_sign`. This happens multiple times in one loop iteration, causing extra overhead shown in Figure 8.

`471.omnetpp` and `483.xalancbmk`. The main source of the overhead is frequent virtual function call, much more than other C++ benchmarks. For every virtual function, PACTIGHT-CPI calls `pct_auth` to authenticate the virtual function pointer and `pct_sign` to re-add the PAC.

Comparison to prior work. PACTIGHT-CPI's overhead in these three benchmarks is similar to the original CPI [38]. Note that PACTIGHT-CPI has double the instrumentation, since it needs to authenticate and then resign, whilst CPI would only compare with its metadata with a single instrumentation. PARTS [43] does not evaluate SPEC CPU2006. Even though PTAuth [27] does evaluate with a subset of the SPEC CPU2006 benchmarks, they do not mention the performance numbers for cc453.povray, `471.omnetpp` and `483.xalancbmk`. We expect that these numbers would be quite high. PACStack [42] evaluates with SPEC CPU2017. Thus, we could only compare definitively with the original CPI.

Possible improvement One of the main reasons overhead is very high is because of instrumentation inside loops. One could say that a possible solution would be hoisting the authenticating and signing to be outside loops, *i.e.*, authenticate before entering a loop and sign after the loop is done. In that way, authenticating and signing would only be done once.