PACTIGHT: Tightly Seal Sensitive Pointers with Pointer Authentication

Mohannad A Ismail

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

December 2, 2021

Blacksburg, Virginia

Keywords: System Security, ARM, Security Policy

PACTIGHT: Tightly Seal Sensitive Pointers with Pointer Authentication

Mohannad A Ismail

(ABSTRACT)

ARM is becoming more popular in desktops and data centers. This opens a new realm in terms of security attacks against ARM, increasing the importance of having an effective and efficient defense mechanism for ARM. ARM has released *Pointer Authentication*, a new hardware security feature that is intended to ensure pointer integrity with cryptographic primitives. Recently, it has been found to be vulnerable.

In this thesis, we utilize Pointer Authentication to build a novel scheme to completely prevent any misuse of security-sensitive pointers. We propose PACTight to tightly seal these pointers from attacks targeting Pointer Authentication itself as well as from control-flow hijacks. PACTight utilizes a strong and unique modifier that addresses the current issues with PAC and its implementations. We implement four defenses by fully integrating with the LLVM compiler toolchain. Through a robust and systemic security and performance evaluation, we show that PACTight defenses are more efficient and secure than their counterparts. We evaluated PACTight on 30 different applications, including NGINX web server and using real PAC instructions, with an average performance and memory overhead of 4.28% and 23.2% respectively even when enforcing its strongest defense. As far as we know, PACTight is the first defense mechanism to demonstrate effectiveness and efficiency with real PAC instructions.

PACTIGHT: Tightly Seal Sensitive Pointers with Pointer Authentication

Mohannad A Ismail

(GENERAL AUDIENCE ABSTRACT)

ARM is slowly but surely establishing itself in the market for desktops and data centers. Intel has been the dominant force for some time but ARM's entrance into that realm opens up new avenues and possibilities for security attacks against ARM machines. Thus, it is becoming increasingly important to develop an effective and efficient defense mechanism for ARM against possible security threats, particularly against memory corruption vulnerabilities. Memory corruption vulnerabilities are still very prevalent in today's security realm and have been for the past thirty years. Different hardware vendors have developed a variety of hardware features to combat them and ARM is no different. ARM has released *Pointer Authentication*, a new hardware security feature that is intended to ensure pointer integrity with cryptographic primitives. Pointer Authentication allows developers to utilize the unused bits of a pointer and add a cryptographic hash that can ensure the pointer hasn't been tampered with. Pointer Authentication has been utilized in other solutions by security researchers. However, these solutions are either incomplete in their coverage or lack enough randomness for the cryptographic hash. In this thesis we utilize Pointer Authentication to build a novel scheme to completely prevent any misuse of security-sensitive pointers in memory corruption attacks. This thesis presents PACTight to tightly seal these pointers from attacks abusing the limited randomness of the hash as well as control-flow hijack attacks. PACTight implements four defenses by fully integrating with the LLVM compiler toolchain. Through a robust and systemic security and performance evaluation, this thesis show that PACTight defenses are more efficient and secure than their counterparts.

Dedication

To my Mom and Dad.

For their endless sacrfice and unwavering love.

Acknowledgments

In the name of Allah the Most Gracious and the Most Merciful. I would like to start off by thanking my advisor, Dr. Changwoo Min, a professor in The Bradley Department of Electrical and Computer Engineering at Virginia Tech. He is a great mentor and adviser. His guidance, help and patience have been invaluable to me and I wouldn't be where I am without his continuous support. He has always been there for me and I never hesitated to ask him any questions. Thank you Dr. Min!

Thank you to Dr. Daphne Yao and Dr. Haining Wang for their service as my thesis committee members and for their valuable feedback and review on this thesis.

A very special thank you to Dr. Yeongjin Jang and Andrew Quach from Oregon State University for their valuable contribution to this work. Their help has been invaluable and their expertise contributed greatly to accomplishing this work.

I am also thankful to my colleagues in the security side of the Computer Systems, Memory and OS Security (COSMOSS) Lab. Their contributions to my research helped in completing my thesis.

Thank you also to all the members of the COSMOSS Lab. We shared many great moments over the past few years and it has been a pleasure being with you and living these moments.

Last but not least, I thank my parents, brothers, sister, grandma, and the rest of my extended family for their continuous encouragement and support. This accomplishment would not have been possible without them

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the ARM processor architcture started penetrating into the mainstream desktop [8] and data center market [7, 52, 54], beyond the mobile/embedded markets. This opens a new realm in terms of security attacks against ARM, increasing the importance of having an 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) [5] aims to defend against these attacks by ensuring the program follows its proper control-flow. This is mainly done by generating a valid control-flow graph (CFG) of the program and making the program conform to it. Most CFI techniques are either efficient but imprecise (coarse-grained) or precise but inefficient (fine-grained).

In order to defend against control-flow hijacking attacks efficiently, ARM has introduced several hardware security features. One of these is *Pointer Authentication (PA)* [53], a security feature that is intended to ensure 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 the 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 [25, 40, 41, 42] and deployed [10, 32], we identified that they are still exposed to attacks, such as using a signing gadget to forge PACs [13], and reusing of PACs [41], which may allow attackers to run arbitrary code.

In this thesis, 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) unforegablity: 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, and is not dangling.

Compared to previous PA-based defense mechanisms, PACTight assumes a stronger thread model such that the attacker has both arbitrary read and write capabilities and it provides better coverage by protecting a variety of security-sensitive pointers. 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 focus on implementing PACTight to achieve pointer integrity by protecting all sensitive pointers and provide precise, deterministic memory safety for these sensitive pointers. Sensitive pointers here refers to all code pointers and data pointers that can reach to a code pointer. Protecting these sensitive pointers achieves the balance for reinforced protection, thus achieving protection against control-flow hijacking attacks and providing memory safety for sensitive pointers. We aim to demonstrate effectiveness and practicallity by evaluating with real PAC instructions. As far as we know, PACTight is the first mechanism to do so.

In summary, we make the following contributions:

- We propose PACTight, a novel and efficient approach to tightly seal pointers by making a pointer with PAC is unforgeable, 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, 4.28% and 23.2% respectively making it possible to implement PACTight defenses in the real-world.

Chapter 2

Background and Motivation

In this section, we introduce control-flow hijacking attacks, ARM's pointer authentication mechanism, defenses based on PAC and its 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. In particular, attackers can alter the value of a code pointer (return addresses and function pointers) by corrupting the memory location that stores the pointer to subvert the execution flow of the program [14, 15, 17, 20, 24, 28, 59].

To defeat the attack, defenders must ensure that the program has no single point that can let attacker corrupt code pointers as well as data pointers that refer to code pointers in its recursive memory dereference chain. This is because that the attack has evolved into several sophisticated techniques since its discovery, especially from being able to chain a single code pointer corruption into multiple control-flow hijacking. Return-oriented programming (ROP) [57], jump-oriented programming (JOP) [15], and counterfeit-object oriented programming (COOP) [55] are the techniques that aim to achieve a Turing-complete code execution by chaining returns, indirect call/jumps, and virtual function calls in an object

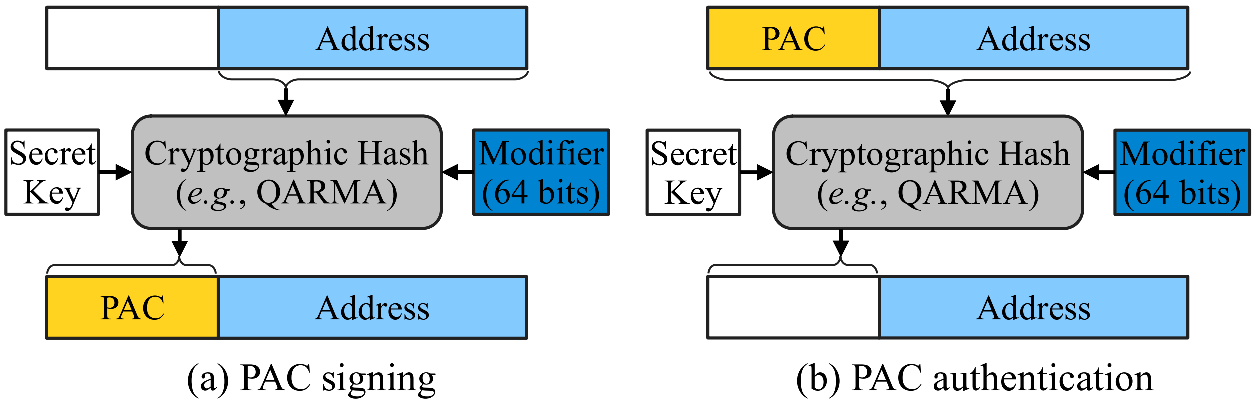


Figure 2.1: PA signs a pointer and generates a pointer authentication code (PAC) based on a address, a secret key, a 64-bit user-provided modifier using PA instructions (*e.g.*, pacia). The signed pointer should be authenticated before the access using the same PAC, address, secret key, and modifier using PA instructions (*e.g.*, autia).

iteration loop, respectively.

2.2 ARM Pointer Authentication

ARMv8.3-A introduced a new hardware security feature called Pointer Authentication (PA). 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. In particular, 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.

PAC signing. PAC utilizes a cryptographic hash algorithm, namely QARMA. The algorithm takes two 64-bit values (the 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. Figure 2.1 shows this signing process. A total of five keys can be chosen to generate the PACs, two keys for code pointers, two keys for data pointers, and one generic key. These are stored in hardware registers, and are secured and protected by the kernel. The keys are the same throughout the lifetime of the process.

PAC authentication. Authentication of the pointer is straight forward. 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 to the one on the pointer, the PAC is removed from the pointer and the pointer can be used, as shown in Figure 2.1b. 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 usually start with either pac or aut followed by a character that identifies whether it's protecting 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. PAC instructions are considered as NOP instructions to have compatibility in older ARM architectures.

PAC has been implemented in the Apple A12 chip [61], and the subsequent ones, as well as recently in the newly announced Apple M1 chip.

2.3 PAC Defense Approaches

Return address focused. Qualcomm's return address signing mechanism [32] 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, aiming to thwart control-flow hijacking attacks originated from overwriting return addresses. This implementation has been supported by the mainline GCC since version 7.0. Apple's implementation of PAC on Clang [10] utilizes a similar scheme for return address protection. Because these two approaches are susceptible to PAC re-use attacks (see section 2.4), PARTS return address protection [41] includes the SP with a function ID, generated by PARTS, as a modiifer 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 the call stack. On the other hand, PCan [40] 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 a variety of other pointers including function pointers and C++ VTable pointers. However, it uses a zero modifier to protect them. PARTS [41] utilizes PAC to protect function pointers, return addresses and data pointers. PARTS mainly relies on static modifiers to sign pointers. It utilizes a type ID based on *LLVM ElementType* as the modifier for signing function pointers and data pointers.

Temporal safety. PTAuth [25] focuses on enforcing temporal memory safety using PAC. To this end, PTAuth generates a new random ID at each memory allocation and utilize 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 such as use-after-free and double-free (also for invalid-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

The PAC itself and defenses based on PAC do not come without their limitations, and they are susceptible to the following attacks.

Forging PAC Attackers may forge a valid PAC for arbitrary pointers if the program logic that generates PAC contains software vulnerabilities. PAC is relying 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 reported attack shows that a memory corruption vulnerability in PAC generation logic may serve as an arbitrary PAC generator, allowing attackers bypassing PAC authentication [13].

Reusing valid PACs in a different context A PAC generated for one context can be reused in a different context if two contexts share a 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 context, such as Qualcomm/Apple stack protection. An example case of the latter is to reuse the PAC generated for a valid return address at 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). Additionaly, PARTS-CFI is also susceptible to this attack because the approach uses a static modifier for the pointer based on its LLVM ElementType. Because the LLVM ElementType can be repeated within the LLVM IR, e.g., 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.

Reusing dangling PACs Attackers can reuse legitimately generated PACs even after the pointer became an invalid, dangling pointer. This case occurs if the modifier used for signing pointer does not convey the temporal status of the pointer. In such a case, the PAC is still valid even after the deallocation of the memory object 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 in Apple's Clang to dynamically check and confirm if the pointers that they protect are not *dangling*, thus they are susceptible to this attack.

Chapter 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 [34, 48]. 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 attacks, such as Spectre [36], are out of scope. Also, data only attacks, which modify and leak non-control data, are out of scope. Our assumptions are consistent with prior works [21, 40, 41, 42] with the exception of PTAuth [25], which aims to defend temporal memory corruption. PTAuth's threat model is weaker, only allowing arbitrary write and not arbitrary read.

Chapter 4

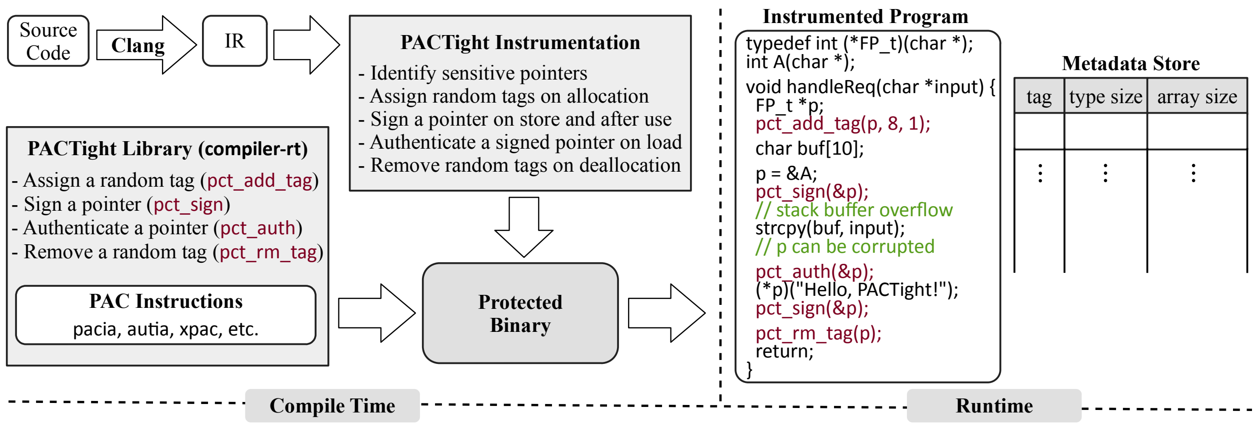
PACTight Design

In this section, we describe the design of PACTight. We first discuss our design goal (section 4.1) and then introduce three pointer integrity properties that PACTight guarantees to overcome the limitations of prior PAC approaches (section 4.2). Lastly, we discuss in detail the design of PACTight. As illustrated in Figure 4.1, PACTight consists of a runtime library and compiler-based instrumentation. We first discuss the runtime library (section 4.3) to explain how PACTight enforces the pointer integrity property and then explain PACTight's automatic instrumentation and defense mechanisms in the next section (Chapter 5).

4.1 PACTight Design Goals

The overarching goal of PACTight is completely preventing 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 section 2.3. In order to achieve our goal, it is essential to enforce the complete integrity of pointers, which we will discuss in section 4.2, and prevent any pointer misuse. We decided to protect sensitive pointers [37] – 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:

- **Pointer integrity**: Prevent any misuse of sensitive pointers.


Source Code
Clang
IR
PACTight Instrumentation
- Identify sensitive pointers
- Assign random tags on allocation
- Sign a pointer on store and after use
- Authenticate a signed pointer on load
- Remove random tags on deallocation
PACTight Library (compiler-rt)
- Assign a random tag (pct_add_tag)
- Sign a pointer (pct_sign)
- Authenticate a pointer (pct_auth)
- Remove a random tag (pct_rm_tag)
PAC Instructions
pacia, autia, xpac, etc.
Protected Binary
Instrumented Program
Metadata Store
tag
type size
array size
Compile Time
Runtime

```
typedef int (*FP_t)(char *);
int A(char *);
void handleReq(char *input) {
  FP_t *p;
  pct_add_tag(p, 8, 1);
  char buf[10];
  p = &A;
  pct_sign(&p);
  // stack buffer overflow
  strcpy(buf, input);
  // p can be corrupted
  pct_auth(&p);
  (*p)("Hello, PACTight!");
  pct_sign(&p);
  pct_rm_tag(p);
  return;
}
```

Figure 4.1: Overall design of PACTight. During the compile time (left), 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 in tandem). PACTight guarantees three pointer integrity properties (section 4.2), namely unforgeability, non-copyability, and non-dangling. During runtime (right), PACTight generates a PAC for sensitive pointers using a novel authentication scheme and checks the PAC upon pointer dereference (section 4.3). PACTight automates its instrumentation in four different levels: forward edge, backward edge, C++ VTable, and sensitive data pointers (Chapter 5).

- **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. Figure 4.2 illustrates the violation of each property, which are discussed in detail below:

- **Unforgeability**: As illustrated in Figure 4.2(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

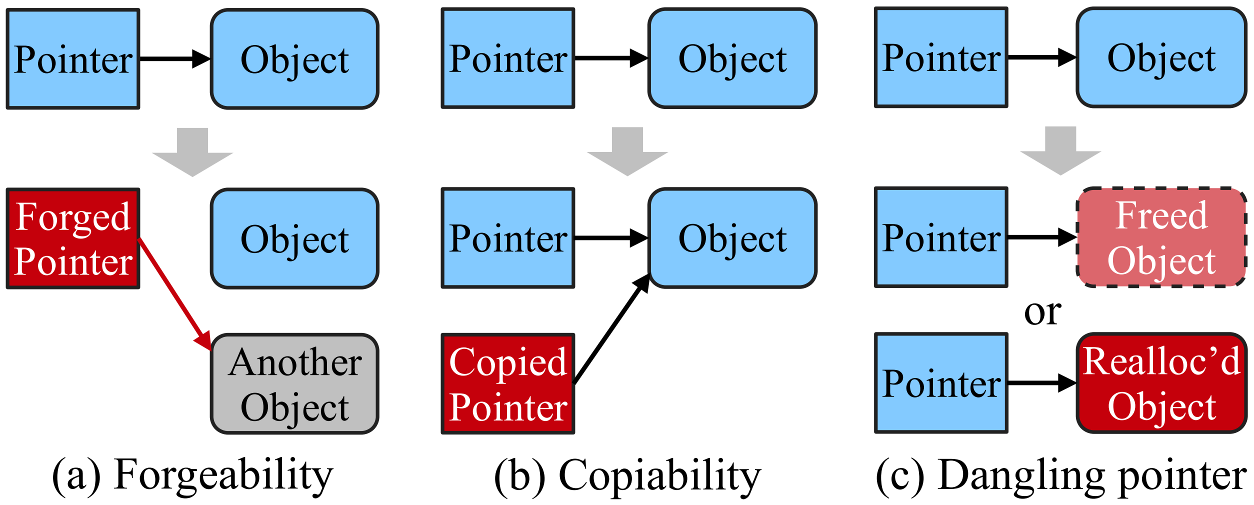


Figure 4.2: Three types of violations of pointer integrity.

memory object and it cannot be altered maliciously.

- **Non-copyability**: A pointer can be copied and re-used maliciously as illustrated in Figure 4.2(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.
- **Non-dangling**: A pointer can refer an unintended memory object if its pointee object is freed or the freed memory is reallocated as illustrated in Figure 4.2(c). In this case, 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 to hijack control-flow. 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 in order to hijack control-flow, at least one of these properties must be violated. PACTight is able to detect any of

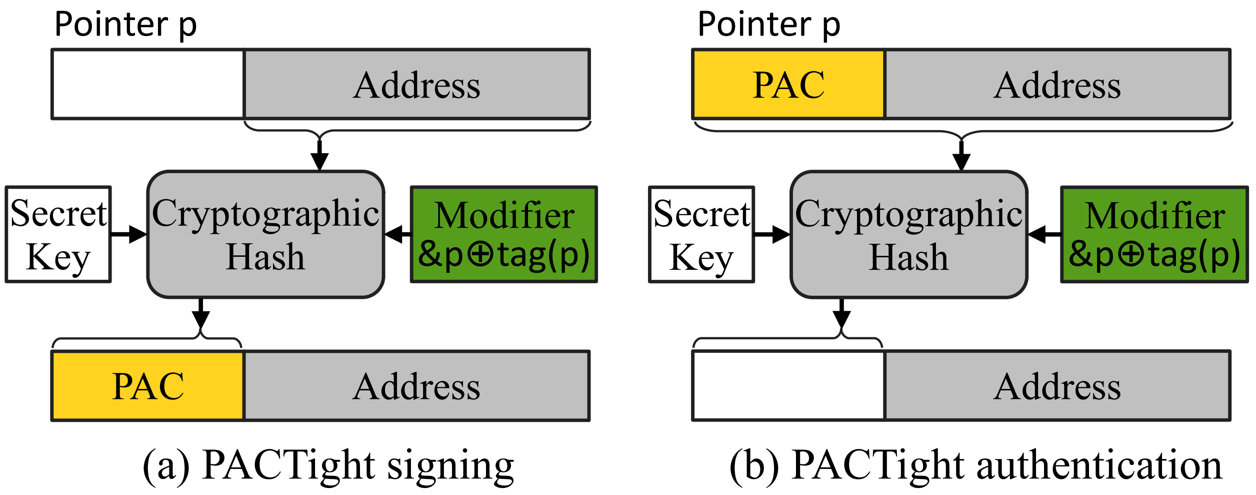


Figure 4.3: Signing and authentication of a pointer variable p in PACTight. In addition to the unforgeability of p provided by PA, PACTight uses the address of a pointer (&p) and a random tag associated with a pointee (tag(p)) to provide the non-copyability and non-dangling properties of a PACTight-signed pointer.

these violations before the use of a pointer, thus guaranteeing the above mentioned pointer integrity. We note that ARM PA only enforces the unforgeability property by signing a pointer with its address (see Figure 2.1).

4.3 PACTight Runtime

This section describe the PACTight runtime. We first describe how PACTight efficiently enforces the pointer integrity properties (subsection 4.3.1), then discuss the PACTight runtime library (subsection 4.3.2) and its metadata store design (subsection 4.3.3).

4.3.1 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 4.3.

- **Unforgeability**: The PAC mechanism includes the pointer as one of the inputs to generate the PAC. If the pointer is forged, it will be detected at authentication. Thus, PAC by itself enforces the unforgeability of a pointer.
- **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 it 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, making all pointers to that memory invalid. Hence, that enforces 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.

4.3.2 Runtime Library

The PACTight runtime library provides four APIs to enforce pointer integrity. The PACTight LLVM instrumentation passes described in Chapter 5 automatically instrument a program using those APIs.

- 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 random tags for each array element by generating 64-bit random numbers. For each element, its associated random tag and size information are added to the metadata store. The API should be called whenever memory is allocated either on the heap or the stack.
- 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, pacib, pacda). If a (compromised) program tries to sign a pointer that does not have an associated random tag – the program tries to access unallocated memory, PACTight aborts the program. This API should be called whenever a pointer is assigned or after the pointer is used.
- pct_auth(&p) 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)) by looking up the metadata store. Then, it authenticates the pointer with the modifier using a PA authentication instruction (*e.g.*, autia, autda, autib). If there is no random tag (*i.e.*, trying to access unallocated memory) or PA fails authentication, PACTight aborts the program. If authentication is successful, it strips off the PAC from the pointer. This API should be called before using the pointer.

- 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.

A running example. The code snippet in Figure 4.1 (right) shows how PACTight APIs are used to protect a local function pointer p. When a pointer gets allocated, pct_add_tag from the runtime library 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. In the case of arrays, each element in the array gets a unique random tag, with the rest of the metadata keeping track of the size of an element and the number of elements in the array. If any change in assignment happens to the pointer legitimately, the PAC is authenticated (pct_auth) and a new PAC is generated for the new pointer with pct_sign. When a pointer gets deallocated, the pointer is authenticated and all metadata is removed (pct_rm_tag). This is done by reading the type size and the array size from the metadata and removing the metadata accordingly. This prevents the metadata from being reused by the attacker.

4.3.3 Metadata Store

PACTight maintains a metadata store for allocated memory objects. For each allocated memory object, the metadata store maintains a 64-bit random tag ID, 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 implemented the metadata store as a hash table using the address (*i.e.*, p) as a key. 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.

Chapter 5

PACTight Defense Mechanisms

In this section, we present the PACTight defense mechanisms built on top of the PACTight runtime. The PACTight compiler passes automatically instrument all globals, stack variables and heap variables in a program, 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 using PACTight's mechanism. 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 legitimate 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.*, looking for BitCast instructions 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.

PACTight-CFI looks for all store instructions. 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 the code pointer. Specifically, PACTight-CFI looks for the relative 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 on a pointer, the PAC should be added again after the function call. Thus, PACTight-CFI instruments pct_sign after indirect call instructions. This ensures that a PAC is always present on the function pointer.

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.

Pointer operations. Since a PACTight-signed pointer has a PAC in its upper bits, care must be taken not to break the semantics of existing C/C++ pointer semantics. In particular, we take care of the following three 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-CFI strips off the PAC before passing the signed pointer as an argument.

Summary. PACTight-CFI is precise by enforcing the PACTight pointer integrity properties and it is efficient by leveraging hardware-based PA. Moreover, it provides the Unique Code Target (UCT) property [31] 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 [39] 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 in the EC.

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 very common exploit in C++ programs [16, 55, 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.

Instrumenting PACTight APIs. Upon a C++ type having a virtual function allocated, PACTight-VTable instruments pct_add_tag similar to PACTight-CFI. 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 the load instruction. A failed authentication flips the top two bits of the pointer, rendering it unusable. pct_sign is instrumented again after the object has been used to seal the pointer

```
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
13
14 /** ==== nginx/http/ngx_http_variables.c ==== */
15 ngx_http_variable_value_t *
16 ngx_http_get_indexed_variable(ngx_http_request_t *r,
17                 ngx_uint_t index) {
18     ngx_http_variable_t  *v; // sensitive pointer
19     // ...
20     v = cmcf->variables.elts; // assignment of v
21     // use of a sensitive point v
22     if (v[index].get_handler(...) == NGX_OK)
23     {
24         // ...
25     }
26     // ...
27 }
```

Figure 5.1: Example of a sensitive data pointer in (simplified) NGINX source code. ngx_http_variable_t is a sensitive data type since it contains a function pointer get_handler (Line 10). Thus, PACTight-CPI instruments its allocation (Line 18), assignment (Line 20), and use (Line 22) to enforce the PACTight pointer integrity properties for sensitive pointers.

again. In the same fashion as setting the metadata, 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 [37]. 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 control-flow by corrupting a sensitive *data* pointer because it can reach a code pointer. Figure 5.1 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 v or its array index index are corrupted between Line 20-22, an attacker can hijack the control-flow at Line 22 without directly corrupting the function pointer, get_handler. PACTight-CPI protects not only sensitive code pointers but also sensitive data pointers for the guaranteed protection against control-flow hijacking.

Identifying sensitive pointers. PACTight-CPI expands the type analysis of PACTight-CFI to include sensitive data pointers as well as code pointer (*i.e.*, all sensitive pointers). It classifies a composite type that contains a function pointer as a sensitive type by investigating all composite types. 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 in a program. When our static analysis is unsure (e.g., C union), we conservatively consider the type as sensitive.

We over-approximate when detecting security-sensitive pointers in our static analysis. That is, we regard a pointer as a security-sensitive pointer if we cannot determine a pointer as non-security-sensitive at static time. This conservative approach may induce false-positives, however, such false positives will not compromise PACTight 's security guarantees but adds additional performance overhead. In our evaluation, these cases were rare and do not occur frequently. In the case of dynamic_cast<T>, dynamic_cast<> 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. 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. At the same time, 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 can not be a dangling pointer.[1] Hence the non-dangling property is not necessary to be enforced so the random tag is also not necessary. Not using the random tag has huge performance benefits because the metadata store lookup 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].

[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 of PACTight.

Signing and authentication of a return address. Our optimized sign/authentication scheme for return address 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-copyabilty* 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. This guarantees that the proper stack semantics are maintained. The LLVM-assigned function ID is unique due to the use of link time optimization (LTO). We further discuss return address protection in Chapter 8.

Chapter 6

Implementation

Module	**Lines of Code**		
	Added	**Deleted**	**Total**
PACTight Library	656	0	656
LLVM Pass	3216	21	3237
AArch64 Backend	109	12	121
Total	3981	33	4014

Table 6.1: Lines of code for the PACTight prototype.

Our prototype consists of around 4000 lines of code (LoC) based on LLVM 10 shown in Table 6.1. PACTight-CFI, PACTight-VTable and PACTight-CPI are all implemented in the LLVM IR level as instrumentation passes while the return address protection is implemented in the AArch64 backend. The PACTight runtime library is integrated with LLVM as part of compiler_rt. PACTight provides compiler flags to enable each defense mechanism discussed in Chapter 5. We implemented the metadata store as a hash table using the memory address (p) as a key. To achieve the best performance possible, we inlined all PACTight runtime library calls and enabled Link Time Optimization (LTO). To further harden our prototype, we used different key types for sensitive function pointers (pacia, autia), sensitive data pointers (pacda, autda), and return addresses (pacib, autib).

Chapter 7

Evaluation

We evaluate PACTight by answering the following questions:

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

7.1 Evaluation Methodology

Evaluation environment. We ran all evaluations on Apple's M1 processor [8], 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 PACTight 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 [10] to avoid the conflicting use of PA instructions.

Evaluation of C++ applications. However, unfortunately we found that the use of PA instructions are built into Apple's standard C++ library so we cannot turn off Apple's PA instrumentation on the library using a compiler flag. We have investigated using the Ubuntu

Linux [19] on the M1 to work around the problem. However, unfortunately the Linux kernel on the 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) [11], which is an ARM hardware platform simulator that supports pointer authentication. We were able to use 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 – CFIXX C++ Test Suite [16] and the seven C++ benchmarks in SPEC 2006 – 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.* [41] also replaced a PA instruction with four eor instructions to estimate the performance overhead, which is more optimistic than our measurement on real hardware.

7.2 Security Evaluation

In this section, we evaluate PACTight's effectiveness in stopping security attacks using three real-world exploits (section 7.2) and five synthesized exploits (section 7.2).

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 [22] in the image file format library libtiff. An attacker triggers an integer overflow using a malicious BMP file, which is then followed by a heap overflow. The heap overflow overwrites a function pointer in the TIFF structure. This vulnerability allows attackers to achieve arbitrary code execution. PACTight-CFI/CPI successfully 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 [23] in libpng [3] library, which implements an interface for reading and writing PNG format files. The exploit is in the png_safe_execute function in the pngerror.c file. The png_image_free_function function is called indirectly and frees memory that is referenced by image. image is then dereferenced afterwards. image – png_image type – has a pointer to the png_control type, which has another pointer to the png_struct type, which has several function pointers in it. Since PACTight-CPI does recursive identification, image is considered sensitive by PACTight-CPI. Thus, when image gets dereferenced after the free, PACTight-CPI will detect that no metadata exists for the pointer and halt the execution.

(3) CVE-2014-1912. This is a buffer overflow vulnerability [56] in python2.7 that happens due to a missing buffer size check. An attacker can corrupt a function pointer in the PyTypeObject with a malicious string and achieve arbitrary code execution. PACTight-CFI/CPI detects this by detecting the corrupted function pointer with pct_auth.

Synthesized Exploits

We evaluated PACTight with five synthesized attacks for C++ to demonstrate how PACTight-VTable can defend against virtual function pointer hijacking attacks, COOP attacks [55] - a Turing complete attack that crafts fake C++ objects. We used CFIXX C++ test suite [50] by Burow *et al.* [16]. 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 [35]. In a nutshell, the virtual function hijacking exploits overwrite a virtual function pointer in the C++ object. PACTight-VTable detects all the exploits by enforcing pct_auth on the virtual function pointer before the virtual function call to detect if it has been corrupted. 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.

7.3 Performance Evaluation

Benchmark applications. For performance evaluation, we use three benchmarks – namely SPEC CPU2006 [30], nbench [46], and CoreMark [2] – which have been used in prior works, and one real-world application, NGINX web server [4]. In order to run SPEC CPU2006, we ported the SPEC benchmark to the Apple M1 and built it from scratch. We were not able to run one benchmark, 403.gcc, in the SPEC benchmark on the Apple M1 even with Apple's vanilla Clang/LLVM compiler so we omitted the results of 403.gcc. We suspect that there is 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

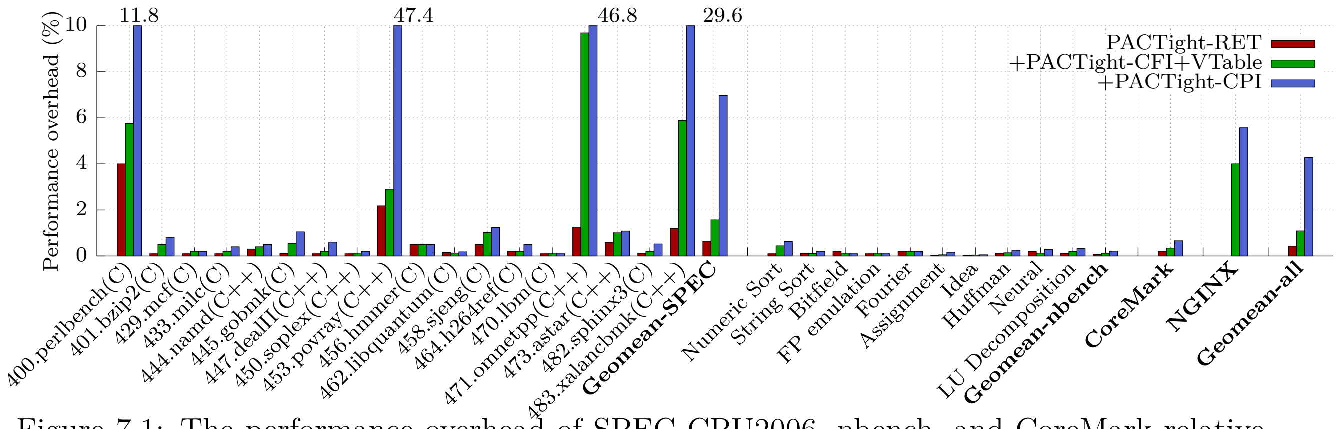


Figure 7.1: The performance overhead of SPEC CPU2006, nbench, and CoreMark relative to an unprotected baseline build. Our three PACTight protections are: 1) return addresses (PACTight-RET), 2) CFI, C++ VTable protection and return addresses (+PACTight-CFI+VTable), and 3) CPI offering full protection for all sensitive pointers (+PACTight-CPI).

C++ benchmarks, we replaced a PA instruction with seven eor instructions to emulate the overhead of the PA instructions as discussed in section 7.1.

Performance overhead. Figure 7.1 shows the performance of the PACTight defenses on the SPEC 2006 benchmarks, nbench, and CoreMark. The SPEC benchmarks have a geometric mean of 0.64%, 1.57%, and 6.97% for PACTight-RET, PACTight-CFI+VTable+RET, and PACTight-CPI, respectively. The geometric means of all benchmark applications are 0.43%, 1.09%, and 4.28% 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 analyzed these three benchmarks, and found that the common culprits for the high overhead are loops with sensitive data pointers in the conditions and variables inside the loop. Specifically, this was due to the recursive nature of authenticating and re-adding the PAC for structures in PACTight-CPI.

We evaluated the NGINX web server on the Apple M1 using its 4 big cores to stress the machine. We used the same configuration F5 Networks used to bench NGINX TLS transac-

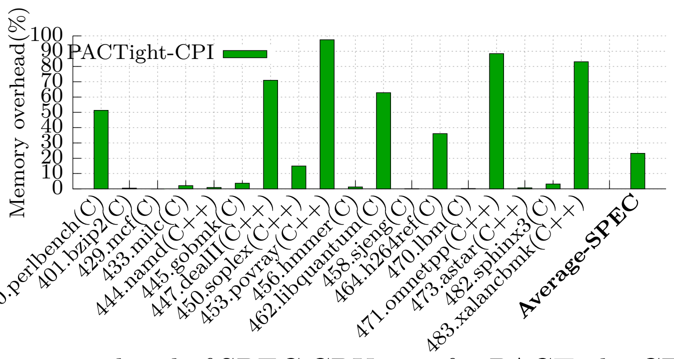


Figure 7.2: The memory overhead of SPEC CPU2006 for PACTight-CPI, PACTight's highest protection mechanism. PACTight imposes a marginally low overhead of 23.2% on average.

tions per second [47]. We used the HTTP benchmarking tool wrk [27] to generate concurrent HTTP requests. wrk starts threads that send requests for a static HTML and measures the throughput (req/sec) and latency. We ran wrk on a different machine under the same network. Each wrk spawns three threads where each thread handling 50 HTTP connections. We observed 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 for PACTight 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 so it requires the largest number of entries in the metadata store. Figure 7.2 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. The overhead is much lower for PACTight-CFI, due to less instrumentations and thus less metadata.

Chapter 8

Discussion and Limitation

Information leakage attack on the metadata store. In our threat model, assuming a powerful attacker with arbitrary read and write capabilities, an attacker is able to access the PACTight metadata store while it is probabilistically hidden using address space layout randomization (ASLR). However, 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. Another hypothetical attack is a case that an attacker reuses a random tag to bypass the non-dangling property. While such attack is possible in theory, the bar is very high in practice; an attacker cannot reuse dangling pointers at 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 Memory Tagging Extension (MTE), ARM's new v8.5 security hardware feature [12]. 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 random tag store.

Alternative Return Address Protection. We believe that extending PACTight's return address mechanism to use MTE or an approach similar to Intel CET [33] – hardware-based secure shadow stack – would result in a more secure mechanism than the current one.

Collision likelihood of PACTight-CFI The likelihood of a collision depends on the number of bits available for the PAC. As discussed in PACStack, an attacker has the ability to notice collisions after it has seen approximately $1.253 \cdot 2^{b/2}$ tokens, where b is the number of bits available for PAC. In our case, b=16 and thus there would be 321 tokens, which is low, however, we use a random tag as a modifier to increase the difficulty in collision. PACStack mitigates this with a random XOR mask, making the modifier more unique and thus forcing 2^{-b} guesses. Compared with PACStack having random XOR, we use a 64-bit random tag which may provide a similar mitigation to the PACStack XOR mask, due to the uniqueness of the modifier.

Chapter 9

Related Work

ARM pointer authentication defenses. PARTS [41] uses PA to enforce pointer integrity for all code and data pointers. It uses a static modifier (type-id) to sign code and data pointers and both dynamic (stack pointer) and static modifiers (function-id) to sign return addresses. This provides fine-grained forward-edge and backward-edge CFI protection along with data pointer protection.

PA has also been applied to strengthen stack protections. PCan [40] dynamically generates stack canaries using PA with the stack pointer and the function-id as modifiers. PAC-Stack [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.

PTAuth [25] provides temporal safety under the assumption the attacker has no arbitrary read. On allocation, it inlines a randomly generated ID for all heap objects and uses it with the object's base address as the PA modifier. On dereference, a backwards search is performed to find the base address of the object. Only heap-based temporal errors are covered by PTAuth.

The related defense mechanisms using PA have weaker guarantees compared to PACTight. They are incomplete under our threat model. PACTight can enforce the three security properties to seal pointers under a powerful threat model without relying on other defense mechanisms whilst achieving low performance and memory overhead. Also, PACTight is the only one of all the currently existing PA mechanisms to utilize real PAC instructions in its evaluation as far as we know.

Cryptographic pointer defenses. CCFI [45] uses MACs to protect return addresses, function pointers, and vtable pointers. Registers are reserved to prevent the key from spilling. Function pointers use the hash of the type and the address of the pointer as modifiers. Return addresses use the old frame pointer as the modifier. Vtable pointers are protected with the address stored. 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) [5] restricts the valid target sites for indirect control-flow transfers. Static CFI schemes are vulnerable to control-flow bending [18]. Adding dynamic checks is needed to reduce the number of valid target sites. Since PACTight-CFI-VTable seals a pointer with its location and a random tag, this limits the feasibility of a reuse attack. Other non-PA dynamic approaches require additional threads to analyze the data from Intel Processor Trace [26, 29, 31, 44] limiting scalability.

Code Pointer Integrity (CPI) [37] 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 the safe stack. PACTight-CPI provides temporal safety to sensitive pointers, which CPI doesn't, and protects virtual function pointers in addition to sensitive pointers, all while having a lower overhead across all defenses.

CFIXX [16] protects VTable pointers by enforcing Object Type Integrity (OTI). CFIXX stores metadata on the construction of an object and checks the metadata at the virtual function call site. CFIXX incurs an average overhead of 4.98%. PACTight-CFI+VTable incurs lower overhead (1.98%) whilst providing stronger guarantees by enforcing CFI.

Hardware-assisted defenses. ARMv8.5 introduces the Memory Tagging Extension (MTE) [12] feature providing spatial and temporal safety for pointer accesses. MTE has a 4-bit tag per every 16 bytes of memory (tag granule). Enabling the tagging feature reduces the number of bits available for PA by eight bits. Intel Memory Protection Extensions (MPX) [51] is a feature providing spatial safety with bounds checking.

Temporal memory safety. Explicit pointer invalidation is a common strategy to enforce temporal memory safety. DangNull [38], DangSan [60], FreeSentry [62], pSweeper [43], and BOGO [64] invalidate all pointers to an object when the object is freed. These schemes typically incur high costs. CRCount [58] implicitly invalidates pointers by using reference counting. This approach comes at memory costs since a dangling pointer will cause objects to never to be freed. CETS [49] uses disjoint metadata to check if an object still exists upon pointer dereferences. MarkUs [6] quarantines freed data and prevents reallocation until there are no dangling pointers pointing to it. MarkUs is a memory allocator that strictly protects against use-after-free attacks only, whilst PACTight is a compiler that protects sensitive pointers from memory corruption attacks.

Chapter 10

Conclusion

This thesis presented PACTight, an efficient and robust mechanism to guarantee pointer integrity utilizing ARM's Pointer Authentication mechanism. We identified three security properties that 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 its pointee object is freed. We implemenented PACTight with four defense mechanisms, protecting forward edge, backward edge, virtual function pointers, and sensitive pointers. We demonstrated the security of PACTight against real-world and syntesized attacks and showcased its low performance and memory overhead, 4.28% and 23.2% respectively, against a variety of benchmarks, including SPEC 2006, nbench and CoreMark with real PAC instructions.

Bibliography

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

[5] 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.

[6] 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.

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

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

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[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] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

[49] 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.

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

[51] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[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.