# Effective and Practical Defenses Against Memory Corruption and Transient Execution Attacks with Hardware Security Features

Mohannad A. Ismail

Preliminary Exam

Doctor of Philosophy

in

Computer Engineering

Changwoo Min, Co-chair

Wenjie Xiong, Co-chair

Danfeng (Daphne) Yao

Haining Wang

Michael Hsiao

September 15, 2023

Blacksburg, Virginia

Keywords: Compiler-enabled Defenses, Practical Design, System Software, Control-flow Hijacking, Data Oriented Attacks, Hardware Security Features

# Effective and Practical Defenses Against Memory Corruption and Transient Execution Attacks with Hardware Security Features

Mohannad A. Ismail

(ABSTRACT)

Memory corruption attacks have existed in C and C++ for more than 30 years, and over the years many defenses have been proposed. However, with every new defense a new attack emerges, and then a new defense is proposed. This is an ongoing cycle between attackers and defenders.

There exists many defenses for many different attack avenues. However, many suffer from either practicality or effectiveness issues, and security researchers need to balance out their compromises. Recently, many hardware vendors, such as Intel and ARM, have realized the extent of the issue of memory corruption attacks and have developed hardware security mechanisms that can be utilized to defend against these attacks. ARM, in particular, has released a mechanism called Pointer Authentication in which its main intended use is to protect the integrity of pointers by generating a Pointer Authentication Code (PAC) using a cryptographic hash function, as a Message Authentication Code (MAC), and placing it on the top unused bits of a 64-bit pointer. Placing the PAC on the top unused bits of the pointer changes its semantics and the pointer cannot be used unless it is properly authenticated.

Hardware security features such as PAC are merely mechanisms not full fledged defences, and their effectiveness and practicality depends on how they are being utililzed. Naive use of these defenses doesn't alleviate the issues that exist in many state-of-the-art software defenses. The design of the defense that utilizes these hardware security features needs to have practicality and effectiveness in mind. Having both practicality and effectiveness is now a possible reality with these new hardware security features.

This thesis proposal describes utilizing hardware security features, namely ARM PAC, to build effective and practical defense mechanisms. This thesis proposal first describes my past work called PACTight, a PAC based defense mechanism that defends against control-flow hijacking attacks. PACTight defines 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) noncopyability: 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. PACTight protects all sensitive pointers, which are code pointers and pointers that point to code pointers. This completely prevents control-flow hijacking attacks, all while having low performance overhead.

In addition to that, this thesis proposal proposes Scope-Type Integrity (STI), a new defense policy that enforces pointers to conform to the programmer's intended manner, by utilizing scope, type, and permission information. STI collects information offline about the type, scope, and permission (read/write) of every pointer in the program. This information can then be used at runtime to ensure that pointers comply with their intended purpose. This allows STI to defeat advanced pointer attacks since these attacks typically violate either the scope, type, or permission. We present Runtime Scope-Type Integrity (RSTI). RSTI leverages ARM Pointer Authentication (PA) to generate Pointer Authentication Codes (PACs), based on the information from STI, and place these PACs at the top bits of the pointer. At runtime, the PACs are then checked to ensure pointer usage complies with STI. RSTI overcomes two drawbacks that were present in PACTight: 1) PACTight relied on a large external metadata for protection, whereas RSTI uses very little metadata. 2) PACTight only protected a subset of pointers, whereas RSTI protects all pointers in a program. RSTI has large coverage with relatively low overhead.

Finally, this thesis proposal explains post-preliminary exam work with hardware security features, namely utilizing hardware security features to defend against speculative execution attacks. The proposed work includes developing a novel defense utilizing ARM pointer authentication to defend against specific variants of speculative execution effectively and practically.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Control-flow hijacking and data-oriented attacks have become more sophisticated in recent years. These types of attacks are extremely dangerous, due to the fact that they can lead to arbitrary code execution, giving an attacker full control of the system. In order to combat these attacks, many security defense mechanisms have been proposed, each enforcing their own protection mechanism. These include data integrity [61, 69, 103], memory safety [33, 37, 40, 86, 115, 122], and control-flow integrity [8, 49, 58, 63]. The main problem in most of these solutions stems from limited coverage, low security guarantees or high performance overhead. Due to the fact there are many moving parts that need to be protected, defenders have to choose what they want to protect and how it will be protected.

The following sections introduce control-flow hijacking and data oriented attacks, the challenges of the current state-of-the-art defenses, and what the goals of this thesis are.

### 1.1.1 Control-flow hijacking and data-oriented attacks

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) [8] 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. Most CFI techniques are either efficient but imprecise (coarse-grained) or precise but inefficient (fine-grained).

On the other hand for data-oriented attacks, attacks such as DOP [57] and NEWTON [113] exploit data pointers to leak sensitive data, such as SSL keys, or achieve arbitrary code execution. These attacks bypass many state-of-the-art defense mechanisms [8, 23, 32, 41, 49, 52, 53, 58, 64, 69, 77, 89, 90, 92, 93, 107, 108, 111, 112, 119, 121]. Such data-oriented attacks are more difficult to defend against as data pointers are much more abundant in programs than code pointers. It is also not easy to distinguish between security-sensitive data pointers (e.g., pointing to an SSL key) and non-security-sensitive data pointers.

### 1.1.2 Challenges in Current State-of-the-art

Whilst current state-of-the-art defense mechanisms have become more sophisticated as the attacks have evolved, there are still many challenges that come up when desigining a defense mechanism to defend against memory corruption attacks.

- **Low coverage** Some current defense mechanisms such as PTAuth [48], PACStack [75] and most CFI defense mechanisms [8, 23, 32, 41, 49, 52, 53, 58, 64, 77, 89, 90, 92, 93, 107, 108, 111, 112, 119, 121] limit themselves in terms of coverage. PACStack

only protects return addresses, and PTAuth only provides temporal memory safety. A practical and efficient defense mechanism needs to have more coverage, as well as more variants of defense mechanisms. More variants allow the defense mechanism to be tailored to different defense needs.

- **Less security guarantees** Whilst a defense mechanism may provide a new defense technique, some current defenses open new avenues for attacker to continue to be able to attack the program or even completely bypass the defense mechanism. Code Pointer Integrity (CPI) [69] protects sensitive pointers (code pointers and pointers that refer to code pointers) by storing the sensitive pointers in a seperate memory region. Whilst the mechanism is effective in preventing control-flow hijacking attacks, the prevalence of the external metadata in memory allows the attacker to completely bypass the mechanism [46], thus negating its effectiveness.
- **High performance overhead** A major problem with many current defense mechanisms is the amount of performance overhead added. Techniques such as CETS [87] and DangSan [109] have average overheads that exceed 100%. A defense mechanism needs to be practical in order to provide justification for its use in real-world applications.
- **Compatibility** Since the main goal is to protect legacy C/C++ programs from attackers, new defense mechanisms must continue to be compatible and not break any C/C++ semantics. Moreover, the defense mechanisms shouldn't ideally need the developer to annotate the program with semantics that are foreign to C/C++. YARRA [97] provides protection to data pointers in a program based on specific types. However, the programmer needs to annotate the variables in the program to enforce the protection. This adds extra non-negligible effort to protect programs, and may cause errors due to wrong annotations that could happen by mistake from the

programmer.

### 1.1.3 Goals

The challenges described above have caused many defense mechanisms to make compromises and choose a specific defense avenue. Moreover, these challenges make it difficult to adopt any of the current state-of-the-art defenses in practice.

This thesis proposal aims to address the following questions:

- Is it possible to design an effective and practical defense mechanism to defend against memory corruption attacks by addressing the challenges above? If so, how?

In order to answer this question, it is essential to investigate why many state-of-the-art defenses lack in effectiveness and practicality. Most state-of-the-art defenses have been implemented as purely software solutions, and this means that sacrifices need to be made when making design choices. Either sacrifice security guarantees and coverage for the sake of better performance, or sacrifice performance for the sake of better coverage and higher security guarantees. In addition to that, the design choices of current defense mechanisms show that there still isn't a clear winning strategy to designing an effective and practical defense mechanism.

This thesis proposal lays out the work for this winning strategy. The key is to utilize the new hardware security features that CPU vendors have implemented, along with smart design choices to be able to achieve effective and practical defense mechanisms against memory corruption attacks. This proposal discusses past work, in terms of design, implementation, and evaluation, as well as current work, also in terms of design, implementation and evaluation, that showcase practical and effective defense mechanisms against memory corruption attacks.

The past work is PACTight, a Pointer Authentication based defense mechanism that enforces three security properties for pointers and completely prevents control-flow hijacking attacks. The current work is Scope Type Integrity (STI), which is the main focus of this proposal, is a new defense policy that enforces that pointers conform to the programmer's intended manner. This work also explains Runtime Scope Type Integrity (RSTI), which leverages Pointer Authentication to enforce STI.

## 1.2 Contributions

The main goal of this thesis proposal is to propose effective and practical defense mechanisms using hardware security features. Specifically, this proposal aims to showcase that it is very much possible to implement effective and practical defense mechanisms with smart design choices and utilizing hardware security features. The following four axioms need to be achieved.

- **High coverage and versatility** This thesis proposal aims to design solutions that covers a wide array of memory corruption attacks, specifically a wide variety of pointers. Specifically, this thesis proposal not only aims to protect as many pointers as possible, but also to provide different layers of coverage that suit different security needs. This showcases both high coverage and versatility of the proposed defense mechansims.
- **High security guarantees** Protecting many pointers and providing many defense mechanisms is futile if the defense mechanisms can be compromised or if the attacker can compromise them. The defense mechanisms proposed aim to significantly raise the bar for attackers to execute their attacks, or even outright making some attacks completely impossible. This means that the existence of the defense mechanism

greatly hinders the attacker's ability.

- **Low performance overhead** Defense mechanisms cannot just be secure, they also must be performant. This thesis proposal aims to introduce smart design choices that greatly reduce the performance overhead and makes it possible for these defense mechanisms to be deployed in real-life. The performance overhead of the defense mechanisms in this thesis proposal is lower than their counterparts in the state-of-the-art.
- **Compatibility** Te main goal is to protect legacy C/C++ programs from attackers, thus new defense mechanisms must continue to be compatible and not break any C/C++ semantics. The proposed defense mechanisms rely on the LLVM compiler infrastructure to automatically instrument the code and guarantee that the programs conform to C/C++ standards.

This thesis proposal discusses a past fully implemented, evaluated and published project, PACTight, that showcases that utilizing hardware security features allows us to achieve the goals discussed above. However, in spite of PACTight's increased coverage there are still some avenues that attackers can use that would be costly if PACTight was to cover them. Thus, this thesis proposal presents the present fully implemented, evaluated and to be published project, RSTI, that overcomes the limitiations of PACTight and covers much more attack avenues. And finally, both PACTight and RSTI together cover all pointer integrity attack avenues, however they do not cover speculative attacks, which have emerged as a serious threat recently. Thus, this thesis proposal proposes a future project that would utilize hardware security features to design a practical and effective defense against speculative execution attacks.

Therefore, this thesis proposal makes the following contributions:

- Discussion of the design, implementation and evaluation of PACTight, a defense mech-

anism that utilizes Pointer Authentication to completely defend against control-flow hijacking attacks. PACTight is implemented as a custom LLVM compiler, and is fully evaluated in terms of its security and performance. PACTight has low performance overhead while offering high security guarantees against control-flow hijacking attacks.

- The design, implementation and evaluation of RSTI, a defense mechanism that protects all pointers in a program from all attacks that violate pointer integrity, such as control-flow hijacking attacks and data oriented attacks. RSTI relies on the scope, type and permission information of variables to bring back the programmer's intent to the runtime. RSTI has relatively low performance overhead and offers high security guarantees against all pointer-based attacks.

## 1.3 Summary of Post-Preliminary Exam Work

After the preliminary exam, this proposes the primary future work to be designing a new defense mechanism that defends against speculative execution attacks by utilizing hardware security features. This defense mechanism will cover speculative execution attacks that can be protected in software without modifying the hardware. Currently we are in the discussion phase of the design.

This thesis also proposes to implement and evaluate the new defense mechanism. A full and comprehensive security and performance evaluation will be done to showcase the value of utilizing hardware security features. This would allow it to be compared with other speculative execution defenses in the literature.

## 1.4 Thesis Organization

This thesis proposal is organized as follows. Chapter 2 presents background information regarding the thesis proposal. This includes more detailed information on the relevant attacks, as well as the hardware defense mechanism, ARM Pointer Authentication. Section 2.4 presents the prior work PACTight, a Pointer Authentication based defense mechanism that enforces three properties of pointers that completely prevent control-flow hijacking attacks. Chapter 3 presents Scope Type Integrity (STI), a new defense policy that brings the programmer's intent to the runtime, as well as its implementation Runtime Scope Type Integrity (RSTI), a Pointer Authentication implementation that enforces STI. Chapter 4 shows the post-preliminary exam work, which includes designing a new defense mechanism that utilizes hardware security features to protect against more advanced speculative execution memory corruption attacks. Chapter 5 concludes and summarizes the past, present and future works showcased in this thesis proposal.

# Chapter 2

# Background

This chapter goes over the background necessary to understand this thesis proposal. A major part of this thesis proposal is about understanding the different attacks and defenses against memory corruption, as well as understanding the ARM Pointer authentication hardware security feature. Section 2.1 goes over the different types of attacks, including control-flow hijacking, data oriented attacks, and speculative execution memory corruption attacks. Section 2.2 explains in detail ARM's Pointer Authentication security mechanism. This is important since it is utilized extensively in the defense mechanisms discussed in this thesis proposal. Section 2.3 briefly goes over some of the current defenses and state-of-the-art.

## 2.1 Memory corruption attacks

### 2.1.1 Control-Flow Hijacking Attacks

The basis of most systems is written in unsafe C and C++ languages that are prone to many memory corruption vulnerabilities, due to the inexistence of sanity checks [106]. Thus, an

```
1  int TIFFWriteScanline(TIFF* tif, ...){
2    ...
3    // Function pointer dereference
4    // Arbitrary address
5    // Execute attack!!
6    status = (*tif->tif_encoderow)(tif, (uint8*) buf,
7                 tif->tif_scanlinesize, sample); }
8  void _TIFFSetDefaultCompressionState(TIFF* tif){
9    // Function pointer assignment
10   tif->tif_encoderow = _TIFFNoRowEncode; }
11 TIFF* TIFFOpen(...){
12   ...
13   _TIFFSetDefaultCompressionState(tif);}
14 int main(int argc, char* argv[]){
15   TIFF *out = NULL;
16   out = TIFFOpen(outfilename, "w");
17   ...
18   uint32 uncompr_size;
19   unsigned char *uncomprbuf;
20   ...
21   uncompr_size = width * length;
22   // Unsanitized Code - Buffer overflow
23   uncomprbuf = (unsigned char *)_TIFFmalloc(
24                          uncompr_size);
25   ...
26   if (TIFFWriteScanline(out, ...) < 0) {}
27   ...}
```

**Figure 2.1:** Control-flow hijacking attack example. The attacker can exploit the buffer overflow vulnerability in Line 21.

attacker may exploit the language semantics to corrupt the memory, and even craft their own attacks with calculated corruptions. One of the critical attack that stem from the memory corruption vulnerabilities is 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 [20, 21, 26, 38, 47, 51, 102].

The control-flow hijacking attack in Figure 3.1 shows a vulnerability (CVE-2015-8668) in the libtiff library. This is a heap-based buffer overflow. The program does not sanitize

```
1 int serveconnection(int sockfd) {
2   char *ptr;
3   ...
4   if (strstr(ptr, "/..")) // Reject the request
5   log( ... ); // Buffer overflow!
6   ...
7   if (strstr(ptr, "cgi-bin") // Handle CGI request
8   ... }
```

**Figure 2.2:** Data-oriented attack example. The attacker corrupts ptr in Line 4 to bypass the security checks.

the buffer size in Line 21. This means that uncomprbuf can be too small, allowing the attacker to overflow heap memory. A possible target is tif_encoderow, which is called by TIFFWriteScanline. The attacker can then overwrite tif_encoderow with an arbitrary address that they can jump to when the function pointer gets dereferenced.

### 2.1.2 Data-Oriented Attack

Data-oriented attacks aim to corrupt non-control data pointers in order to maliciously leak information [30, 57] or achieve arbitrary code execution. These attacks are much more powerful than control-flow hijacking attacks, due to the fact that they do not touch any control data. They have been used, for example, to leak an SSL key [44, 57]. They are harder to defend against due to the abundance of data pointers in a program and the inability to distinguish security-sensitive data pointers from non-security-sensitive ones.

Figure 3.2 shows an attack against the GHTTPD web server [31]. In this attack, the attacker relies on corrupting the pointer ptr. They send an HTTP request with a crafted URL. Then, they trigger the buffer overflow vulnerability in log() and overwrite the address in pointer ptr to the address of the crafted URL. This crafted URL allows the attacker to bypass the input validation checks in Lines 4 and 8, and the attacker executes /bin/sh. Manipulating data pointers is often the desired attack vector for data-oriented attacks [35].

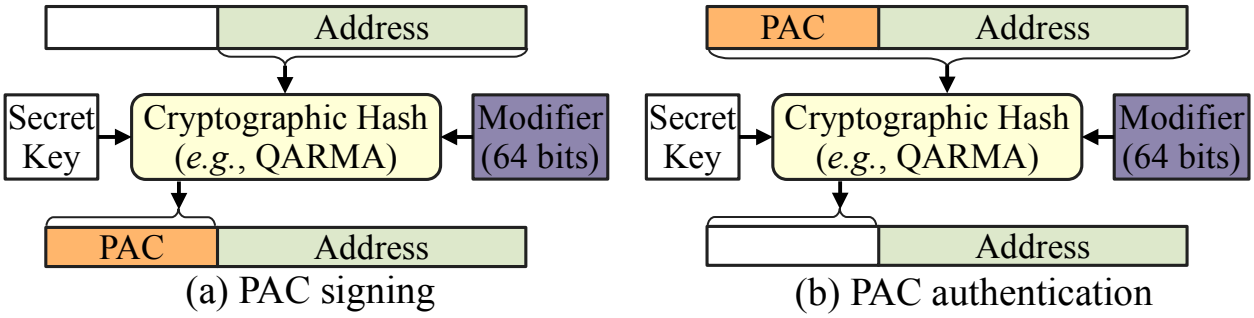



**Figure 2.3:** PA signs a pointer and generates a pointer authentication code (PAC) based on a address, a secret key, a 64-bit user-provided modifier using PA instructions (*e.g.*, pacia). The signed pointer should be authenticated before the access using the same PAC, address, secret key, and modifier using PA instructions (*e.g.*, autia).

## 2.2 ARM Pointer authentication

ARMv8.3-A [59] introduced a new hardware security feature, Pointer Authentication (PA). PA has been implemented in the Apple A12 and M1 chips [114]. Recently, ARM announced ARMv8.6 with Enhanced PAC [105]. This iteration of PAC features several improvements. Instead of flipping the top two bits of the pointer upon authentication failure, an exception is thrown, immediately terminating the program. In addition to that the PAC is XORed with the top bits of the pointer rather than just placed. At the moment, there is no publicly available hardware with ARMv8.6 support. The goal of PA is to protect the integrity of security-critical pointers, such as code pointers. The specification introduces new instructions that can be utilized for creating (sign) and authenticating (verify) PACs, which are in essence a Message Authentication Code (MAC) generated as a cryptographic hash of the pointer value, a key, and a 64-bit modifier. 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.

Using MACs to enforce pointer integrity isn't new. CCFI [81] utilizes MACs to enforce CFI

for control-flow data such as function pointers, return addresses and VTable pointers. CCFI uses a hardware implementation of AES to calculate the MACs. However, software checks are still necessary to enforce its protection mechansim. PAC is the hardware realization of CCFI and with the checks done in hardware, it is much faster. The security of a cryptographic scheme relies on its ability to prevent the attacker from abusing its hash, and thus the uniqueness of the modifier is of utmost importance.

**PAC signing.** PAC utilizes a cryptographic hash algorithm, namely QARMA [17]. The algorithm takes two 64-bit values (pointer and modifier), as well as a 128-bit key, and generates a 64-bit PAC. It is not necessary to use the QARMA algorithm. A manufacturer can use a manufacturer-specific cryptographic algorithm of their own. These PACs are truncated and added to the upper unused bits of the 64-bit pointer as illustrated in Figure 3.3(a). Five keys in total can be chosen to generate the PACs, two for code pointers, two for data pointers, and one generic key. These keys are stored in special hardware registers, which 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 3.3(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.

## 2.3 Current Defenses

### 2.3.1 PAC Defense Approaches

PAC has great potential in terms of enforcing fine-grained pointer integrity, especially against control-flow hijacking attacks. It's hardware implementation of the cryptography and placing the PAC in the unused bits of the pointer with a single instruction mean that it's potential enforcement overhead should be minimal. Usage of the unused bits of the pointer means that the PAC can be propagated with the pointer, and thus there is no extra storage for using PAC.

**Return address focused.** Qualcomm's return address signing mechanism [59] 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. Thus, any overwrite of the return address to initiate a ROP attack would be thwarted. This implementation has been supported by the mainline GCC since version 7.0. Apple's implementation of PAC on Clang [13] utilizes a similar scheme for return address protection.

However, because this approach is susceptible to PAC re-use attacks, PARTS return address protection [74] includes the SP with a function ID, generated by PARTS, as a modifier

to harden the PAC scheme against re-use attacks. Moreover, PACStack [75] extends the modifier by chaining PACs to bind all previous return addresses in a call stack. On the other hand, PCan [73] 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 [74] 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 with PAC.** PTAuth [48] 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. This ID is stored at the beginning of the object and is used as a modifier to compute a PAC for the object pointer.

### 2.3.2 Other Defense Approaches

**Integrity policies.** Control-Flow Integrity (CFI) [8] restricts the valid target sites for indirect control-flow transfers. Static CFI schemes are vulnerable to control-flow bending [27]. Other non-PA dynamic approaches require additional threads to analyze data from Intel Processor Trace [49, 53, 58, 77] limiting scalability.

Code Pointer Integrity (CPI) [69] 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.

CFIXX [24] 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 average overhead of 4.98%.

**Type-based defenses.** The goal of EffectiveSan [43] is to do bounds checking by combining type checking with low fat pointers. TDI [85] relies on grouping types into specific memory arenas by relying on a special allocator and compiler instrumentation. Type after type [110] replaces regular allocations with typed allocations that never reuse memory.

**Cryptographic pointer defenses.** CCFI [81] 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.

**Temporal memory safety.** Explicit pointer invalidation is a common strategy to enforce temporal memory safety. DangNull [70], DangSan [109], FreeSentry [118], pSweeper [76], and BOGO [122] invalidate all pointers to an object when the object is freed. These schemes typically incur high costs. CRCount [101] implicitly invalidates pointers by using reference counting. This approach comes at memory costs since some objects may never be freed. CETS [87] uses disjoint metadata to check if an object still exists upon pointer dereferences. MarkUs [9] is a memory allocator that protects from use-after-free attacks. It quarantines freed data and prevents reallocation until there are no dangling pointers.

## 2.4 Past work: PACTight

### 2.4.1 Introduction

In recent years, the ARM processor architcture started penetrating into the data center [10, 91, 94] and mainstream desktop [11] 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) [8] 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)* [59], 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 [48, 73, 74, 75] and deployed [13, 59], we identified that they are still exposed to attacks, such as using a signing gadget to forge PACs [18] and reusing PACs [74], 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 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, 4.07% and 23.2% respectively making it possible to deploy PACTight defenses in the real-world.

### 2.4.2 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 [62, 84]. 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 [67], and data only attacks, which modify and leak non-control data, are out of scope. Our assumptions are consistent with prior works [39, 73, 74, 75] with the exception of PTAuth [48], which only allows arbitrary write and not arbitrary read.

### 2.4.3 PACTight Design

In this section, we describe the design of PACTight. We first discuss our design goal (Section 2.4.3), then we introduce three pointer integrity properties that PACTight enforces to overcome the limitations of prior PAC approaches (Section 2.4.3), and then we compare PACTight to current state-of-the-art defenses (Section 2.4.3). Lastly, we present

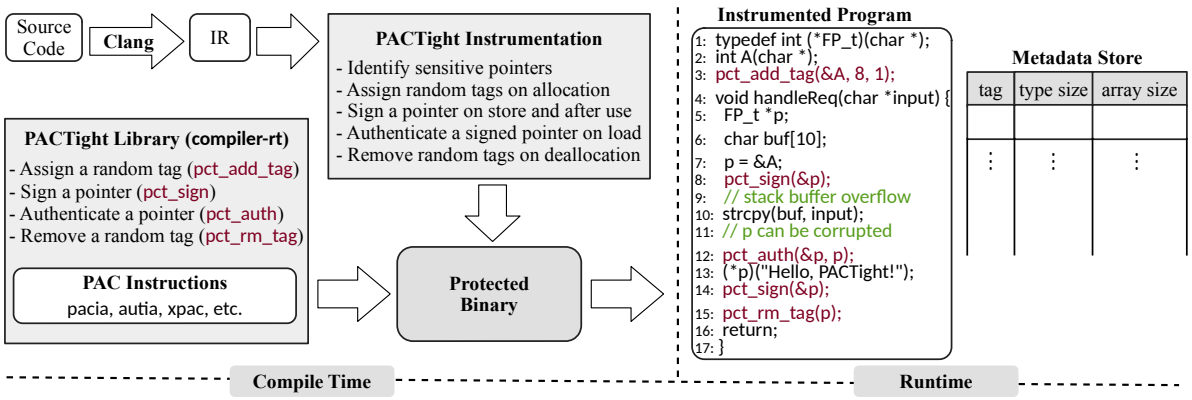



**Figure 2.4:** PACTIGHT design. At compile time, PACTIGHT instruments the allocation, assignment, use, and deallocation of code pointers and data pointers that are reachable to a code pointer (*i.e.*, sensitive pointers). PACTIGHT guarantees three pointer integrity properties (Section 2.4.3), namely 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 (Section 2.4.3). PACTIGHT automates its instrumentation in four different levels: forward edge, backward edge, C++ VTable, and sensitive pointers (§ 2.4.4).

the detailed design of PACTIGHT. As shown in Figure 2.4, PACTIGHT consists of a runtime library and compiler-based instrumentation. We first discuss the runtime (Section 2.4.3) to explain how PACTIGHT enforces the pointer integrity properties and then explain PACTIGHT's automatic instrumentation and defense mechanisms (§ 2.4.4).

### 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. To achieve our goal, it is essential to enforce the complete integrity of pointers, which we will discuss in Section 2.4.3, and prevent any pointer misuse. We protect sensitive pointers [69] – 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:

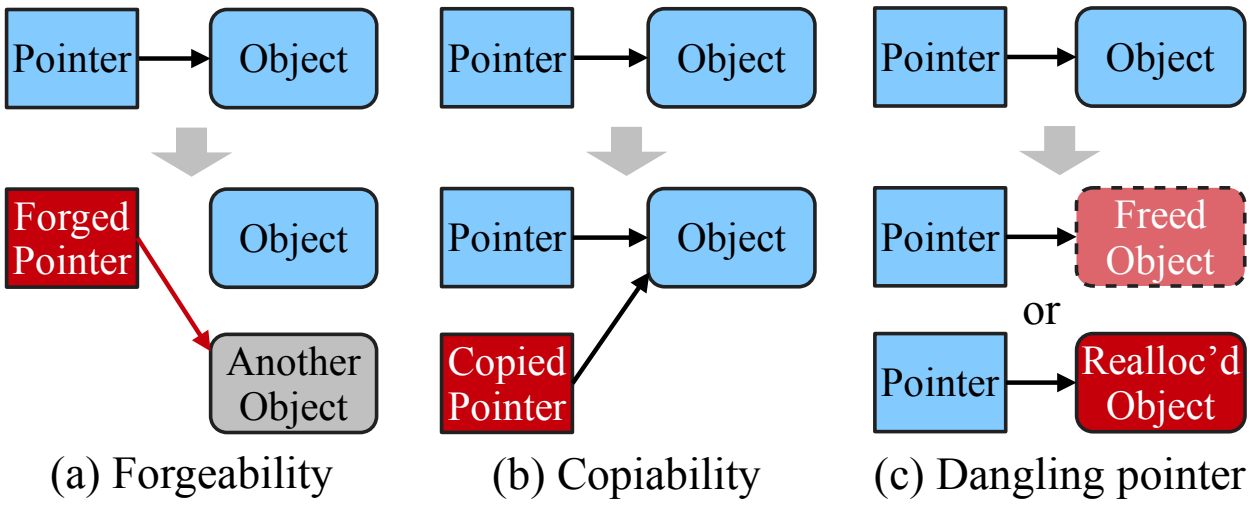



**Figure 2.5:** Three types of violations of pointer integrity.

- **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.

### 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 2.5(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 2.5(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 2.5(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.

### Comparison against Other PAC-based Defenses

In contrast to other PAC-based defenses, PACTIGHT offers more coverage against PAC attacks. PARTS [74] 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 [75] introduces the idea of cryptographically binding a return address to a partic-

ular 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 [48] attempts to provide protection against 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.

### PACTight Runtime

This section describes the PACTight runtime. We first describe how PACTight efficiently enforces the pointer integrity properties, then discuss the PACTight runtime library, pointer operations and the metadata store design .

**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 2.6.

- **Unforgeability**: PAC by itself enforces the unforgeability of a pointer. PAC includes the

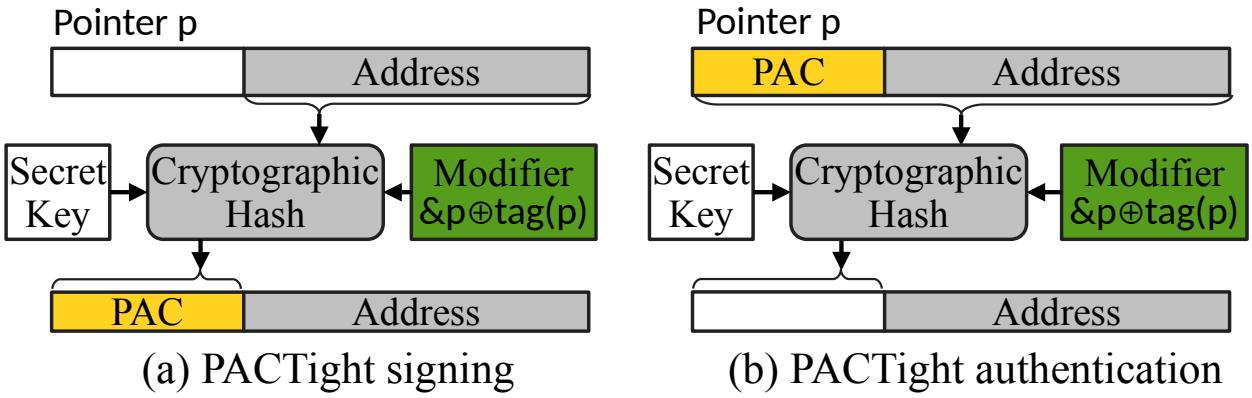



**Figure 2.6:** Signing and authentication of a pointer variable p in PACTIGHT. In addition to the unforgeability of p provided by PA, PACTIGHT uses the address of a pointer (&p) and a random tag associated with a pointee (tag(p)) to provide the non-copyability and non-dangling properties.

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, 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.

**Runtime Library** The PACTight runtime library provides four APIs to enforce pointer integrity. The PACTight LLVM instrumentation passes described in § 2.4.4 automatically instrument a program using those APIs.

**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 **??** (see Figure 2.7). 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.

**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 2.7 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.

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 [48] performs "backward search" to find an array base address, which is not necessary in PACTight.

**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 **??**.

We implemented the metadata store as a linear open addressing hash table (base + offset)

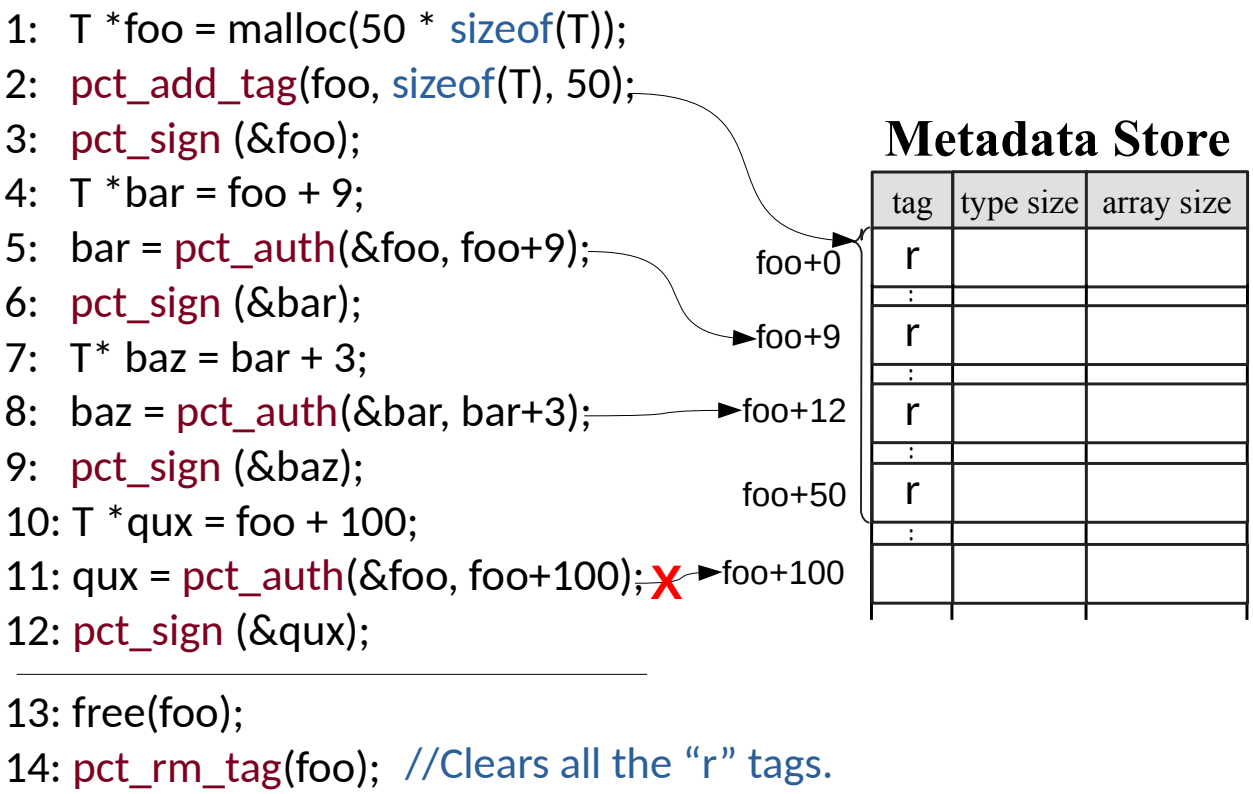



**Figure 2.7:** Example of PACTIGHT handling array operations.

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.

**A Running Example** The code snippet in Figure 2.4 (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.

### 2.4.4 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).

#### 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 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.*, 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 [58] 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 [71] 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.

### 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 [24, 99, 120].

**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

```
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 2.8:** Example of a sensitive data pointer in the (simplified) NGINX source code.

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 loading 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).

### Code Pointer Integrity (PACTight-CPI)

PACTIGHT-CPI increases the coverage of PACTIGHT-CFI to guarantee integrity of all sensitive pointers [69]. 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 2.8 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.

### 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 [75]. 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

[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.

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).

#### Optimization to Reduce PAC Instructions

The main source of overhead in PACTight would be due to the cryptographic operations done by the QARMA algorithm. This is done every time a PAC instruction is executed. As discussed in **??**, 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.

### 2.4.5 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. PACTight provides compiler flags to enable each defense mechanism discussed in § 2.4.4. We use CSPRNG seeded by hardware RNG (RNDR in ARMv8) [2] for random tag generation. 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).

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 Section 2.4.4, to reduce PAC instructions. The evaluation of the impact of these optimizations is discussed in **??**.

### 2.4.6 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? (§ 3.6.1)
- How much performance and memory overhead does PACTIGHT impose? (§ 3.6.3)

#### Evaluation Methodology

**Evaluation environment.** We ran all evaluations on Apple's M1 processor [11], which is the only commercially available processor supporting ARMv8.4 architecture with ARM PA instructions. Specifically, we used an Apple Mac Mini M1 [12] 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 [13] 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 [34] 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) [14], 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.* [74] also replaced a PA instruction with four eor instructions to estimate the performance overhead, which is more optimistic than our measurement on hardware.

### Security Evaluation

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

**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 [42] 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 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 [45] in the libpng [4] 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 [100] 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.

**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 [99] – an attack that crafts fake C++ objects. We used CFIXX C++ test suite [88] by Burow *et al.* [24]. 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 [64]. 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.

```
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 2.9:** Example of vulnerable code that PACTight defends against but PARTS [74] cannot.

**Vulnerable code to other PAC defenses.** We describe here a synthesized exploit that bypasses PARTS [74] and PTAuth [48], 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 2.9. PARTS [74] 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.

### Performance Evaluation

### Benchmarks

**Benchmark applications.** For our performance evaluation, we use three benchmarks – namely SPEC CPU2006 [55], nbench [82], and CoreMark [3] – which have been used in prior works, and one real-world application, NGINX web server [7]. In order to run the SPEC

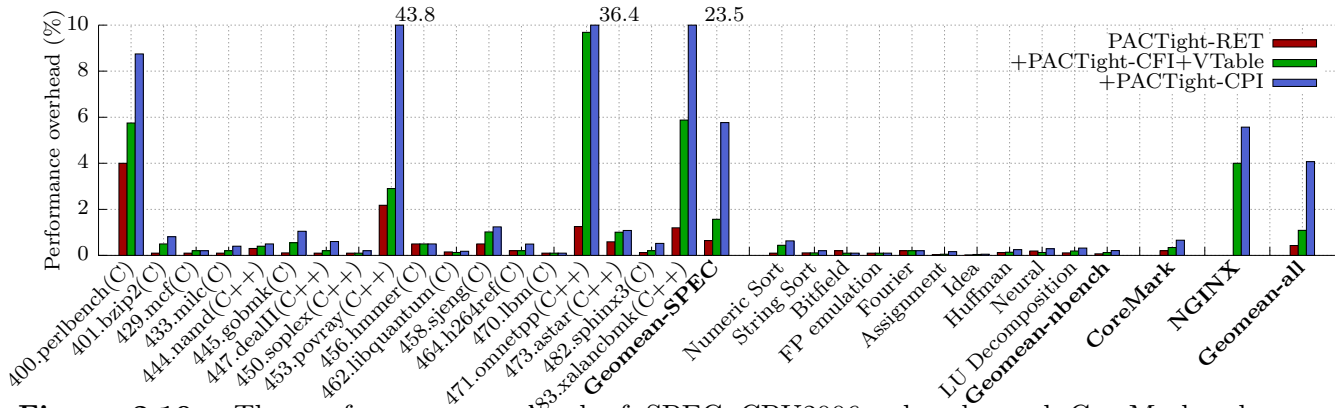



**Figure 2.10:** The performance overhead of SPEC CPU2006, nbench, and CoreMark relative to an unprotected baseline build. Our three PACTIGHT protections are: 1) return addresses (PACTIGHT-RET), 2) CFI, C++ VTable protection and return addresses (+PACTIGHT-CFI+VTable), and 3) CPI offering full protection for all sensitive pointers (+PACTIGHT-CPI).

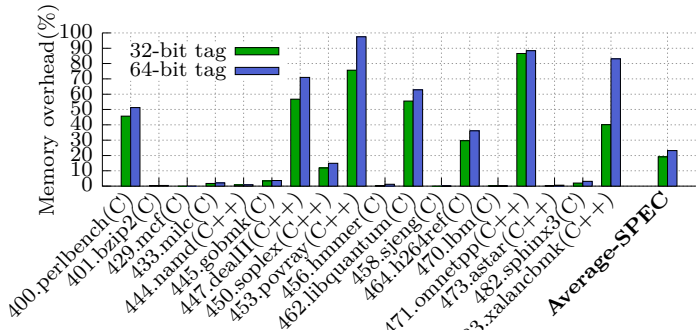



**Figure 2.11:** The memory overhead of SPEC CPU2006 for PACTIGHT-CPI PACTIGHT's highest protection mechanism. PACTIGHT imposes a low overhead of 23.2% and 19.1% on average for 64-bit and 32-bit tags, respectively.

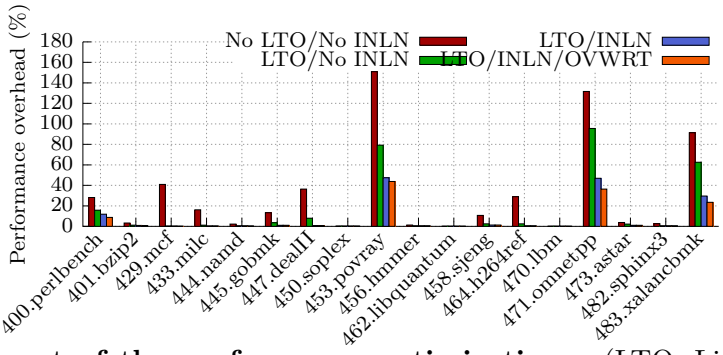



**Figure 2.12: Impact of the performance optimizations.** (LTO: Link Time Optimization INLN: inlining APIs; OVWRT: overwrite stripped pointer with PACed pointer)

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 Section 3.6.3.

**Performance overhead.** Figure 3.9 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 evaluated NGINX on the Apple M1 using its 4 big cores to stress the machine. We used the same configuration used to bench NGINX TLS transactions per second [83]. We used the HTTP benchmarking tool wrk [50] 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 2.11 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× the size of the pointer (64-bit tag) or 1.5× the size of the pointer (32-bit tag).

**Impact of Optimizations** Here we showcase the impact of the optimizations discussed in Section 2.4.4 and Section 3.5. We added three optimizations to PACTIGHT to improve performance: Link Time Optimization (LTO), inlining of PACTIGHT runtime library function (INLN), and overwriting the stripped pointer with a PACed pointer (OVWRT). Figure 2.12 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.

### 2.4.7 Discussion and Limitations

**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). One might wonder what would happen if the metadata gets leaked. 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 attack is a case 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 Memory Tagging Extension (MTE), ARM's new v8.5 security hardware feature [16]. 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.

# Chapter 3

# RSTI: Enforcing C/C++ Type and Scope at Runtime for Control-Flow and Data-Flow Integrity

## 3.1 Introduction

Control-flow hijacking and data-oriented attacks have become more sophisticated in recent years. For example, attacks such as DOP [57] and NEWTON [113] exploit data pointers to leak sensitive data, such as SSL keys, or achieve arbitrary code execution. These attacks bypass many state-of-the-art defense mechanisms [8, 23, 32, 41, 49, 52, 53, 58, 64, 69, 77, 89, 90, 92, 93, 107, 108, 111, 112, 119, 121]. Such data-oriented attacks are more difficult to defend against as data pointers are much more abundant in programs than code pointers. It is also not easy to distinguish between security-sensitive data pointers (e.g., pointing to an SSL key) and non-security-sensitive data pointers. These attacks abuse pointers in unintended ways, far from what was meant by the programmer.

Several defense mechanisms have been proposed to protect data pointers [28, 35, 54, 97]. However, they suffer from reliance on programmer annotation, large dynamic metadata, partial protection (protecting only a subset of data pointers), and/or high overhead. Reliance on programmer annotation [98] makes it impractical to protect legacy programs. Moreover, the programmer may not accurately annotate everything, resulting in more possible bugs. Large dynamic metadata [69] can be abused by an attacker to bypass defenses that heavily rely on metadata for enforcement. More importantly, a creative attacker may abuse other non-instrumented pointers to bypass the defense mechanism if the program is only partially instrumented.

During a control-flow hijacking or data-oriented attack, pointers are corrupted at runtime and the program behaves anomalously, *i.e.*in a way that is not intended by the programmer. In C/C++ programs, variables have a specific type, are valid inside a specific scope, and have specific permissions. These are all restrictions imposed by the programmer. However, at runtime, these restrictions are lost and are not enforced in machine code. Compared to prior work, our design philosophy is to leverage the restrictions inherently defined in the source code at runtime, so that the program can be executed in the proper programmer-intended manner, leaving little room for attackers to manipulate the program.

To this end, this paper proposes *Scope-Type Integrity (STI)*, a new defense policy that enforces pointers to conform to the programmer's intended manner, by utilizing scope, type, and permission information. STI collects information offline about the type, scope, and permission (read/write) of every pointer in the program. This information can then be used at runtime to ensure that pointers comply with their intended purpose. This allows STI to defeat advanced pointer attacks since these attacks typically violate either the scope, type, or permission.

We present *Runtime Scope-Type Integrity (RSTI)*. RSTI leverages ARM Pointer Authenti-

cation (PA) to generate Pointer Authentication Codes (PACs), based on the information from STI, and place these PACs at the top bits of the pointer. At runtime, the PACs are then checked to ensure pointer usage complies with STI. In this way, pointers are guaranteed to execute in the manner specified by the programmer in the application. This allows integrity checks to be performed without needing much external metadata and provides high-security guarantees against creative attacks. Also, leveraging ARM PA allows runtime checks to be efficient, thus opening the way to efficiently and securely protect all pointers in a program. We introduce RSTI-STWC (Scope-Type without Combining), which is our main RSTI mechanism. RSTI-STWC protects all pointers in a program and re-signs pointers whenever a cast happens so that type semantics of the program can still be conformed to. We further introduce two variants of RSTI-STWC. RSTI-STC (Scope-Type with Combining), a relaxed version of RSTI-STWC, and RSTI-STL (Scope-Type with Location), a stricter version of RSTI-STWC. With these three mechanisms, we are able to show the variety of trade-offs between security and performance of RSTI. To this end, we make the following contributions:

- We introduce a new defense policy, Scope-Type Integrity (STI), that enforces a pointer to conform to the programmer's intent. It utilizes scope, type, and permission information to be used after it has been lost during compilation. To our knowledge, STI is the first of its kind to defend against pointer-based attacks by hardening the programmer's intent into machine code.
- We propose Runtime Scope-Type Integrity (RSTI), an efficient enforcement of STI using ARM's Pointer Authentication (PA). We introduce three RSTI mechanisms, RSTI-STWC, RSTI-STC, and RSTI-STL, that instrument all pointers with different restrictions, showing a trade-off between security guarantees and performance overhead.
- We prototype all three RSTI mechanisms on the LLVM compiler, and demonstrate the

```
1  int TIFFWriteScanline(TIFF* tif, ...){
2    ...
3    // Function pointer dereference
4    // Arbitrary address
5    // Execute attack!!
6    status = (*tif->tif_encoderow)(tif, (uint8*) buf,
7                 tif->tif_scanlinesize, sample); }
8  void _TIFFSetDefaultCompressionState(TIFF* tif){
9    // Function pointer assignment
10   tif->tif_encoderow = _TIFFNoRowEncode; }
11 TIFF* TIFFOpen(...){
12   ...
13   _TIFFSetDefaultCompressionState(tif);}
14 int main(int argc, char* argv[]){
15   TIFF *out = NULL;
16   out = TIFFOpen(outfilename, "w");
17   ...
18   uint32 uncompr_size;
19   unsigned char *uncomprbuf;
20   ...
21   uncompr_size = width * length;
22   // Unsanitized Code - Buffer overflow
23   uncomprbuf = (unsigned char *)_TIFFmalloc(
24                          uncompr_size);
25   ...
26   if (TIFFWriteScanline(out, ...) < 0) {}
27   ...}
```

**Figure 3.1:** Control-flow hijacking attack example. The attacker can exploit the buffer overflow vulnerability in Line 21.

feasibility of the proposed schemes on a commercial processor.

- We provide a comprehensive security evaluation of RSTI and its mechanisms on state-of-the-art control-flow hijacking and data-oriented attacks, as well as real-world CVEs.
- We evaluate RSTI's mechanisms on a variety of benchmarks and real-world programs, including SPEC 2006, SPEC 2017, nbench, NGINX and CPython PyTorch, with average overheads of 5.29%, 2.97%, and 11.12% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

## 3.2 Background and Motivation

### 3.2.1 Control-flow Hijacking

```
1 int serveconnection(int sockfd) {
2   char *ptr;
3   ...
4   if (strstr(ptr, "/..")) // Reject the request
5   log( ... ); // Buffer overflow!
6   ...
7   if (strstr(ptr, "cgi-bin") // Handle CGI request
8   ... }
```

**Figure 3.2:** Data-oriented attack example. The attacker corrupts ptr in Line 4 to bypass the security checks.

Control-flow hijacking attacks are critical because they may allow attackers to run arbitrary code. A popular way to carry out a control-flow hijacking attack is exploiting memory corruption vulnerabilities, which C/C++ programs are prone to having. In particular, attackers can alter the value of a code pointer (*e.g.*function pointers) by corrupting the memory location that stores the pointer to subvert execution flow of a program [20, 21, 26, 38, 47, 51, 102].

The control-flow hijacking attack in Figure 3.1 shows a vulnerability (CVE-2015-8668) in the libtiff library. This is a heap-based buffer overflow. The program does not sanitize the buffer size in Line 21. This means that uncomprbuf can be too small, allowing the attacker to overflow heap memory. A possible target is tif_encoderow, which is called by TIFFWriteScanline. The attacker can then overwrite tif_encoderow with an arbitrary address that they can jump to when the function pointer gets dereferenced.

### 3.2.2 Data-Oriented Attack

Data-oriented attacks aim to corrupt non-control data pointers in order to maliciously leak information [30, 57] or achieve arbitrary code execution. These attacks are much more powerful than control-flow hijacking attacks, due to the fact that they do not touch any control data. They have been used, for example, to leak an SSL key [44, 57]. They are harder to defend against due to the abundance of data pointers in a program and the inability to distinguish security-sensitive data pointers from non-security-sensitive ones.

Figure 3.2 shows an attack against the GHTTPD web server [31]. In this attack, the attacker relies on corrupting the pointer ptr. They send an HTTP request with a crafted URL. Then, they trigger the buffer overflow vulnerability in log() and overwrite the address in pointer ptr to the address of the crafted URL. This crafted URL allows the attacker to bypass the input validation checks in Lines 4 and 8, and the attacker executes /bin/sh. Manipulating data pointers is often the desired attack vector for data-oriented attacks [35].

### 3.2.3 Scope, Type, and Permission in C/C++

When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:

- **Basic Type**: Each variable must have a specific type, *e.g.*char, int*. This type is defined by the developer to tell the compiler how this variable will be used in the program.
- **Scope**: Scope defines where the variable will be used. For example, in Figure 3.2, the pointer ptr's scope is the function serveconnection, and should not be used outside that.
- **Permission**: We refer to permissions as whether a variable is defined as read or read/write. A programmer usually defines this by using const in the variable definition.

These three properties express what the developer's intent is for the usage of variables. However, such information is lost after compilation. Thus, an attacker can easily weaponize pointers without conforming to their proper usage. STI aims to analyze the program and retrieve this information. Then, this information would be passed to RSTI to enforce the policy at runtime with ARM PA. Note that STI only concerns itself with pointer variables, since this is the variable type that is usually manipulated by attackers.

### 3.2.4 ARM Pointer Authentication

We leverage the Pointer Authentication feature in ARM to enforce RSTI. ARM introduced Pointer Authentication (PA) in ARMv8.3-A [59] and is available in commercial machines. PA is a hardware security feature that aims to protect the integrity of pointers. It does this by generating a Pointer Authentication Code (PAC) with a cryptographic hash algorithm. For signing, the algorithm takes the pointer, a 64-bit modifier, and a secret key. It then generates a PAC that is placed at the top unused bits of the pointer, as shown in Figure 3.3(a). For authentication, the algorithm takes the PAC'ed pointer, as well as the same modifier and key. The PAC is then recalculated and checked with the one on the pointer. If they match, the PAC is removed, as shown in Figure 3.3(b). If they do not, the top two bits of the pointer are flipped, causing the pointer to be unusable. PA has pac instructions for signing pointers and aut instructions for authenticating, with separate instructions depending on the secret key and type of pointer.

We leverage PA in RSTI to enforce scope, type, and permission. RSTI instruments all pointer loads/stores with PA instructions (pac/aut), leveraging LLVM's pointer authentication intrinsics (llvm.ptrauth) [5]. llvm.ptrauth.sign and llvm.ptrauth.auth take three arguments:

- pointer address: This is the raw pointer to be signed.
- key: This is an integer that identifies which key will be used.
- discriminator: The discriminator is the modifier, a 64-bit integer that adds additional diversity to the PAC.

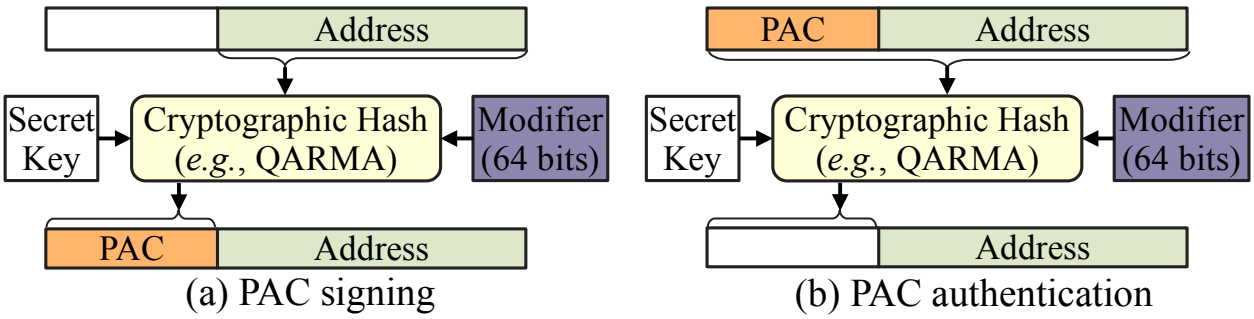



**Figure 3.3:** PA mechanism signs a pointer and produces a Pointer Authentication Code (PAC) based on the pointer, a user-provided modifier, and a secret key.

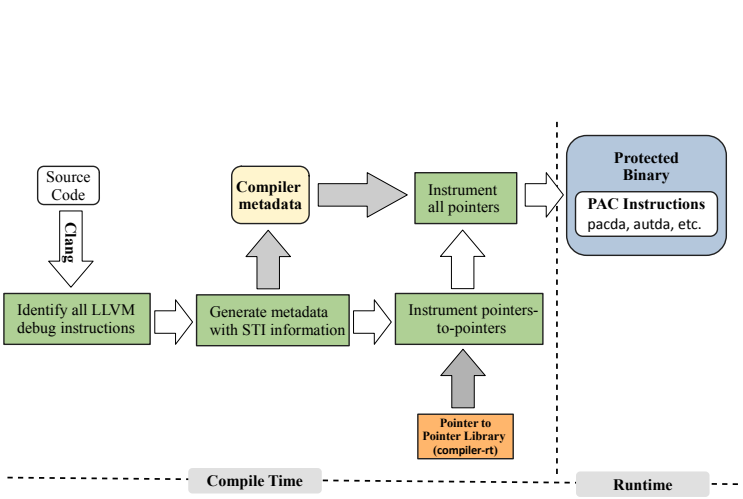
## 3.3 Threat Model and Assumptions

We follow a threat model of typical memory-corruption attacks [106]. The attacker's goal is to achieve arbitrary code execution or memory access by hijacking control/data flow, by abusing a memory corruption vulnerability. RSTI does not prevent corruption of pointers, but rather prevents an attack from achieving code execution stemming from arbitrary read/write that can result from abusing pointers. We assume that Data Execution Prevention (DEP) is in place, and thus the attacker cannot inject their own code. DEP is now enabled by default in most modern operating systems [62, 84]. Also, we trust the hardware and the kernel; more specifically, we trust that PA keys are securely generated, managed, and stored by the kernel. Attacks that target the kernel and hardware, such as transient execution attacks [67, 95], are out of scope.

## 3.4 Runtime Scope-Type Integrity (RSTI)

### 3.4.1 Design Goals

The main goal of RSTI is to protect *all pointers* in a program from memory corruption attacks. This puts RSTI in a position similar to Data Flow Integrity (DFI) [29] and other




```
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

**Figure 3.4:** Overview of RSTI. Starting with the source code, RSTI compiler identifies the scope-type information, generates the metadata, and instruments all pointers. The metadata is static and only used during the compilation phase. The code snippet on the right shows how the scope, type, and permission information is identified by RSTI through the llvm.dbg information.

similar techniques [35, 60, 74]. Some techniques that protect data pointers were limited by their reliance on programmer annotation [97]. Other mitigation techniques [60, 69] rely on external metadata in memory. This causes high overhead, due to the abundance of data pointers, and exposes the metadata to an attacker. Thus, in designing RSTI, we wanted to overcome these challenges. In summary, our main goals are:

- **Completeness:** Protection of all pointers in a program.
- **Little reliance on external metadata:** Eliminate metadata lookup overhead and attacks on the metadata.
- **High performance:** Keep runtime overhead as low as possible.
- **Compatibility**: Allow protection of legacy (C/C++) programs without needing programmer annotations.

### 3.4.2 Design Philosophy

The goal of the defense mechanisms is ensuring correct and proper execution of a program. Meanwhile, the developer of the program has put much information whilst they were writing the program to make sure that it executes in a certain way. All this information is, unfortunately, lost at runtime. So why not use this information as the security context that we need, and propagate it to the runtime in some way? If we can ensure that the program executes in the way that the developer intended even when an attacker is present, then the program would not be compromised, since an attacker relies on the anomalous execution of a program by exploiting vulnerabilities. However, this information needs to be carefully chosen, in order to guarantee enough uniqueness of the security context between different pointers. We identified three main vital pieces of information, that are defined by a developer to execute the program correctly, and these are the *scope*, *type*, and *permission* (defined in § 3.2.3). Below, we show how to defend against a control-flow hijacking attack and data-oriented attack, in Figures 3.1 and 3.2, respectively.

**Defending control-flow hijacking.** For the control-flow hijacking attack in Figure 3.1, if we can enforce that the pointer tif->tif_encoderow conforms to its type (TIFFCodeMethod) and its scope (TIFFWriteScanline and _TIFFSetDefaultCompressionState), then the attacker would not be able to overwrite it with an arbitrary pointer. The attacker wouldn't even be able to use another valid pointer in the program if it does not meet these restrictions.

**Defending data-oriented attacks.** By enforcing scope, type, and permission, the attacker cannot easily overwrite the data pointer ptr in Figure 3.2. The attacker can only corrupt the pointer with an alternative pointer of the same type (char*) and the same scope (serveconnection). This reduces the attack vector significantly.

### 3.4.3 Design Overview

RSTI aims to protect all pointers from being abused by enforcing Scope-Type Integrity (STI) at runtime. STI ensures that pointers are always dereferenced from the correct scope, with the correct type and correct permissions. If an attacker corrupts a pointer, they cannot manipulate the PAC to bypass authentication. This allows RSTI to enforce the programmer's intent without relying on extra annotation.

Figure 3.4 shows the overall design of RSTI. At compile time, all pointers in the source code are instrumented depending on which RSTI mechanism is chosen. The RSTI compiler collects and analyzes all the LLVM debug information to generate compiler metadata. This metadata is then used by LLVM ptrauth intrinsics to generate the protected binary with PAC instructions. At runtime, ARM PA is used to efficiently enforce STI.

### 3.4.4 Scope, Type, and Permission

**Programmer's intent on pointer properties.** STI uses type, scope, and permission information as the programmer's intent to enforce protection. Such information can be collected at compile time. We show that STI is secure enough to ensure that a program executes as the developer intended in § 3.6.1.

- **Basic Type**: Enforcing the type at runtime allows the program to interpret a raw address in the intended way. This is a useful security context since many attackers rely on type-confusion. Type-confusion here means *illegal* type confusion, in which an attacker replaces a pointer of one type with a pointer of another *incorrect* type. By *incorrect* here, we mean against the programmer's intent. If a cast is in the code, it is considered correct from RSTI 's perspective.

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                       int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const =
17                   malloc(sizeof(void));
18   // llvm.ptrauth.sign(v_const,
19   //                              2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   // llvm.ptrauth.sign(c, 2, M2);
22   foo2((void*) c);
23   ... }
```

| Type | Scope | Permission | RSTI Type |
|---|---|---|---|
| ctx* | main,foo, bar,foo2 | R/W | M1 |
| void* | foo2 | R/W | M2 |
| void* | main | R | M3 |

**(a)** RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                       int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const =
17                   malloc(sizeof(void));
18   // llvm.ptrauth.sign(v_const,
19   //                              2, M2);
20
21
22   foo2((void*) c);
23   ... }
```

| Type | Scope | Permission | RSTI Type |
|---|---|---|---|
| ctx*, void* | main,foo2 | R/W | M1 |
| void* | main | R | M2 |

**(b)** RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                       int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const =
17                   malloc(sizeof(void));
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const,
20   //                              2, M3);
21   // llvm.ptrauth.auth(c, 2, M1);
22   foo2((void*) c);
23   ...}
```

| Type | Scope | Permission | RSTI Type |
|---|---|---|---|
| ctx* | main | R/W | M1 |
| void* | foo2 | R/W | M2 |
| void* | main | R | M3 |
| ctx* | foo | R/W | M4 |
| ctx* | bar | R/W | M5 |

**(c)** RSTI-STL

**Figure 3.5:** Code examples with instrumentation. The table below each snippet shows how the RSTI-type is internally stored.

- **Scope**: Enforcing the scope at runtime allows the use of a pointer variable to be bound to a certain subset of program code. For example, a programmer may define a variable void* p to be used in a function foo(). Thus, the scope of p is foo. This specific example is for a local variable. If the variable escapes, *i.e.*it is used in other functions, then the scope is widened to these other functions.
- **Permission**: Enforcing the permission at runtime allows the read-only variables to not be abused by an attacker with read/write variables.

**Extracting scope, type, and permission from LLVM IR.** LLVM's Intermediate Representation (IR) allows us to obtain all of the scope-type information with LLVM's IR metadata. LLVM generates llvm.dbg instructions, that can be used to access the metadata of

the variable. STI uses these instructions to initialize the internal static metadata with the scope, type, and permission information. Figure 3.4 (right) shows a code snippet from a C program and the corresponding LLVM debug metadata. The variable cp is defined with a type of void*. Its scope is the function main. It has read-only permissions, due to the const. The type information is in Line 4 of the LLVM IR, which is the i32* type. The scope information is initially obtained from the instruction !13 by traversing the llvm.dbg instruction, to get to the !DILocalVariable. When instrumenting loads and stores, the scope can be obtained with the !16 instruction and every load/store has one. Thus, this means the appropriate scope can always be obtained. The permission information requires further traversal of the !DILocalVariable to get to the !DIDerivedType. There are multiple layers of !DIDerivedType metadata and we check their tags to find the DW_TAG_const_type tag. This means that this variable is a read-only variable. Scope-type information is then stored in internal compile-time metadata.

**Pointer Casting.** C/C++ semantics allow types to be cast from one type to another. Thus, those two types become *compatible*, since it is explicitly done by the programmer or by the compiler. If a pointer gets cast from one type to another, then we define these types as *compatible types*.

### 3.4.5 RSTI-types

We define *RSTI-type* as a pointer type which restricts a pointer to conform to the scope-type information. If a pointer does not have the intended RSTI-type, RSTI will detect that the pointer has been tampered with. For example, the table in Figure 3.5a shows two RSTI-types (M2 and M3) for one basic type (void*), due to the fact that there are two void* variables that have different scopes and different permissions.

**Distinguishing between local and escaping variables.** RSTI-type also distinguishes between variables that are local to a function and variables that escape. The local variable's scope would only have the one function it is in, and an escaping variable's scope would include all the functions it is being used in, *e.g.*M1 in the table in Figure 3.5a. The function isNonEscapingLocalObject in LLVM helps RSTI with this distinction. This allows RSTI to be precise in its enforcement, whilst maintaining the correctness of the program. We clarify the RSTI-type with examples and explain the different RSTI defense mechanisms in the next section.

### 3.4.6 Enforcement and Defense Mechanisms

RSTI instruments all pointer load/store instructions and enforces RSTI-type with PAC. RSTI instruments all pointers on the stack, heap, and globals. RSTI decides the scope based on where that pointer is legitimately defined and used in the program. The RSTI-type is included in the modifier that is passed to the cryptographic algorithm. Any change in any one of the components means that a different PAC will be generated, and thus will fail on authentication. This mechanism allows STI's restrictions to be enforced at runtime. RSTI supports uninstrumented libraries by stripping the PAC when an external library call is made.

We propose three RSTI defense mechanisms, each with its own distinct view of the RSTI-type. RSTI's three security mechanisms have different security guarantees due to the different nature and strictness each mechanism has to offer.

**RSTI-STWC (Scope-Type Without Combining)** is our main RSTI mechanism. RSTI-STWC authenticates and re-signs pointers when casts happen. For example, in Figure 3.5a, there are two basic types in the program (void* and ctx*). Due to RSTI-STWC's analysis

considering scope and permission, there are now three *RSTI-types*. Even though there is a cast between void* and ctx*, RSTI-STWC maintains separate RSTI-types for them. RSTI-STWC makes sure that legitimate casts are handled appropriately by re-signing the PAC with the RSTI-types after casting, as in Lines 20-22 in Figure 3.5a. However, RSTI-STWC would not be able to detect if two pointers with the same scope-type information are substituted.

**RSTI-STC (Scope-Type with Combining)** combines compatible types together so that it wouldn't need to re-sign pointers when pointer casts happen. For example, in Figure 3.5b, ctx* and void* are combined into *one RSTI-type*, and thus there is no need for additional instrumentation to handle them. The viability of a pointer substitution attack depends on whether the scope-type information of the pointers being substituted are both compatible or not. If they are not compatible, then RSTI-STC can detect the attack. If they are, then RSTI-STC falls short here. RSTI-STC has less security guarantees but offers better performance. This is discussed in detail in Section 3.6.

**RSTI-STL (Scope-Type with Location)** is the strictest RSTI defense mechanism. It combines the location, *i.e.*address, of the pointer with the scope-type information to completely prevent any pointer substitution attacks. In addition to instrumenting casts, RSTI-STL re-signs any pointers that get passed as arguments, due to a change in the location of the pointer. RSTI-STL includes the location of a pointer p (&p) in the modifier. This means that the pointer can only be used legitimately at that specific location [60]. Anytime the location of the pointer changes, RSTI-STL authenticates and re-signs within the new location. This allows it to overcome the shortcomings of the others, albeit at a higher performance cost due to a higher number of instrumentations.

### 3.4.7 Enforcement Details

#### On-Store Pointer Signing

RSTI instruments all pointer stores in a program. They are all signed with their respective RSTI-type as the modifier. Thus, all pointers in a program always have a PAC on them and are always protected. For example, Line 14 in Figure 3.5a.

#### On-Load Pointer Authentication

RSTI authenticates pointers as they are loaded from memory, using the same RSTI-type that was used to sign them on-store. The LLVM pointer authentication intrinsics allow authentication to happen without spilling to memory, due to them being optimized in the compiler. This means that we do not need to re-sign pointers after authentication, since the compiler optimizes the memory accesses and the authenticated address is always in a register. For example, Line 20 in Figure 3.5a.

#### Pointer Operations

Due to the fact that PAC changes pointer semantics, care must be taken when instrumenting pointers in a program.

**Universal pointer types.** Universal pointer types such as void* and char* are abundant in C/C++ programs. RSTI treats them just as it would treat any other type. This allows for consistent instrumentation across programs that may have their own types that are abundant as well.

**Pointer Arithmetic.** RSTI supports pointer arithmetic operations. However, RSTI-

```
1  void hello_func(){printf("Hello!"); }
2  struct node {
3     int key;
4     int (*fp)();
5     struct node *next; };
6  int main(void){
7      struct node* ptr = (struct node*)
8                         malloc(sizeof(struct node));
9      ptr->fp = hello_func;
10     ptr->fp(); }
```

**Figure 3.6:** Composite type example. RSTI handles composite types and enforces the scope of its members to that type.

STWC and RSTI-STC are not capable of enforcing bounds on a pointer. RSTI-STL adds the location of the pointer to the modifier and is able to enforce correct pointer arithmetic without needing bounds.

### Field Sensitive Analysis

RSTI does field-sensitive analysis on composite type variables in order to achieve finer-grained and accurate enforcement of the programmer's intent.

RSTI handles members of a composite type and assigns the scope and type appropriately. Figure 3.6 shows an example of a variable ptr, which is of type struct node* and its scope is main. Its member variable, fp, has a type of int()* but its scope is both main as well as struct node. RSTI enforces that the ptr->fp() has the proper scope in terms of the functions it is executed in, as well as its composite type. Note that composite types can act as both scope and type, depending on which pointers are being referred to.

As for the composite types, LLVM identifies them in the debug metadata. Every struct type has a !DICompositeType in the LLVM IR. This allows RSTI to distinguish composite types from other types that use the !DIDerivedType. Enforcing the scope and type on struct pointer

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                       malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ... }
```

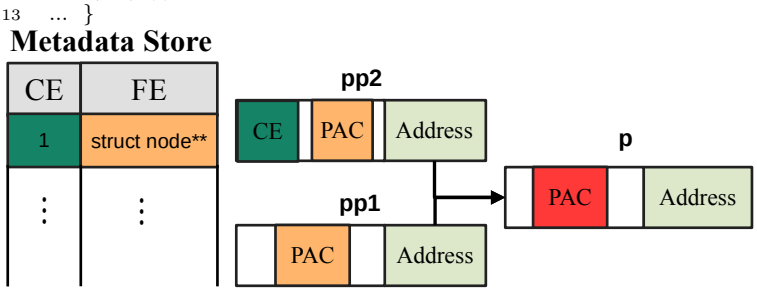



**Figure 3.7:** Pointer-to-pointer handling to preserve the original type. The Compact Equivalent (CE) and Full Equivalent (FE) refer to a tag and the original type of the pointer-to-pointer, respectively.

members is done in two ways. First is using the actual type the member where is defined in. The actual type is represented in the IR. The second is enforcing the struct type where the pointer is a member in. This is done by accessing the GEP (getelementptr) instruction. Struct members are accessed in the IR with the GEP instruction, and this can be used to get the struct type and reach the !DICompositeType to enforce the struct type.

### Pointer-to-Pointer handling

Due to the reliance on the RSTI-type when signing/authenticating pointers, we must make sure that the correct types are being used and propagated. For single pointers, this is not an issue. However, for a pointer-to-pointer, this can be a possible issue, as the original type of the pointer can be lost, especially when the double pointer is casted and passed as a function argument. Thus, we need to find a way to preserve the original type of the pointer in order to make sure that it does not violate the programmer's intent. Our solution is to

store the original type on the pointer itself. However, the number of bits on the pointer that are available (8 bits in case of ARMv8) is limited. We resolve this by adding an extra step of indirection. We define a tag that gets added to the top bits and that leads us to the full original type along with its modifier from the RSTI-type. The tag and the full original type are referred to as the Compact Equivalent (CE) and Full Equivalent (FE), respectively.

Figure 3.7 shows how the mechanism works. Functions foo1 and foo2 take &p, a pointer-to-pointer, as an argument. For foo1, the same basic type (struct node) is in the argument and thus the pointer-to-pointer mechanism is not needed. However, for foo2, the type is void**. In this case, the original type (struct node**) is lost and thus the mechanism is needed. Thus, our pointer-to-pointer solution resolves this by storing a tag (CE) that refers to the original type (FE) on the top 8 bits of the pointer. The CE and FE are also both stored in read-only metadata in memory and can only be accessed by the pointer-to-pointer library in compiler-rt. To implement this, RSTI makes use of an ARMv8 hardware feature called Top Byte Ignore (TBI) [15]. TBI is a feature that ignores the top 8 bits of a virtual address.

**Usage.** Not all pointer-to-pointer types are covered. Only ones that are lost when the pointer type goes out of scope, (*e.g.*, being casted and passed as a function argument) and thus cannot be statically detected. While there are different pointer-to-pointer scenarios, few pointer-to-pointers fall into this specific category. The IR provides sufficient information to handle pointer-to-pointer cases that do not fall into this category. Thus, we do not need to instrument every pointer-to-pointer with this mechanism. Since the CE can only be 8 bits, this means that only 256 types can be used. We evaluate the usage of our pointer-to-pointer mechanism in Section 3.6.2.

**Enforcement.** RSTI checks for all instances where a pointer-to-pointer is being casted and then passed as a function argument. Then, the CE and FE are obtained, added to the table, and the pointer is signed with a PAC based on its RSTI-type and based on the CE.

```
1 void foo(){
2   void *p1, *p2;
3   int* p3;
4   ...
5   p1 = (void*) p3;
6   ...}
```

| | **RSTI-STWC** | **RSTI-STC** | **RSTI-STL** |
|---|---|---|---|
| **p1 and p2** | Merges RSTI-type of p1 and p2 | Merges RSTI-type of p1 and p2 | Doesn't merge |
| **p1 and p3** | Doesn't merge | Merges RSTI-type of p3 with p1 and p2 | Doesn't merge |

**Figure 3.8:** RSTI merging of types. The variation allows for different performance and security guarantees.

When authenticating a pointer-to-pointer, RSTI checks to see if that RSTI-type exists. Then, RSTI uses the CE to obtain the modifier for the original type from the table in memory. In this way, the original type of the pointer can be obtained and authentication can be done. RSTI uses a runtime library to handle pointer-to-pointer. Its functions are:

- pp_add: This is called when a casted double pointer function argument is detected. It adds the FE to the metadata along with its modifier from the internal compiler metadata (Line 9 in Figure 3.7).
- pp_sign: This is called before a store of a casted double pointer function argument. It signs the pointer with a PAC based on the RSTI-type and metadata (Line 10 in Figure 3.7).
- pp_auth: This is called before a load of a signed casted double pointer. It authenticates that pointer with the RSTI-type and the original type obtained from the metadata (Line 3 in Figure 3.7).
- pp_add_tbi: This is called following pp_sign. It adds the CE to the top bits of the pointer so that they can be used to obtain the original type when pp_auth is called (Line 11 in Figure 3.7).

**Table 3.1:** Real and synthesized exploits. This table shows which specific pointers are being abused and how the scope-type information changes.

| Attack Category & Type | | Pointers being abused | Original scope-type information | Corrupted scope-type information |
|---|---|---|---|---|
| **Control flow hijacking** | NEWTON CsCFI attack [113] | **Corrupted:** c->send_chain<br>**Target:** malloc | **Type:** ngx_send_chain_pt<br>**Scope:** ngx_http_write_filter | **Type:** void* (size_t size)<br>**Scope:** libc |
| | AOCR NGINX Attack 1 [96] | **Corrupted:** task->handler<br>**Target:** _IO_new_file_overflow | **Type:** void (*handler) (void *data, ngx_log_t *log)<br>**Scope:** ngx_thread_pool_cycle | **Type:** int *(File *f, int ch)<br>**Scope:** libc |
| | AOCR NGINX Attack 2 [96] | **Corrupted:** p=log->handler<br>**Target:** ngx_master_process_cycle | **Type:** ngx_log_writer_pt<br>**Scope:** ngx_log_set_levels | **Type:** void *(ngx_cycle_t *cycle)<br>**Scope:** main |
| | AOCR Apache Attack [96] | **Corrupted:** eval->errfn<br>**Target:** ap_get_exec_line | **Type:** sed_err_fn_t<br>**Scope:** sed_reset_eval, eval_errf | **Type:** char *(apr_pool_t *p, const char *cmd, char * const *argv)<br>**Scope:** set_bind_password |
| | COOP REC-G [36] | **Corrupted:** objB->unref<br>**Target:** virtual ~Z() | **Type:** class X<br>**Scope:** class Z | **Type:** class Z<br>**Scope:** class Z |
| | COOP ML-G [99] | **Corrupted:** students[i] ->decCourseCount()<br>**Target:** virtual ~Course() | **Type:** void *()<br>**Scope:** class Student, class Course | **Type:** class Course<br>**Scope:** class Course |
| | PittyPat COOP Attack [41] | **Corrupted:** member_2->registration<br>**Target:** member_1->registration | **Type:** void*()<br>**Scope:** main, class Student | **Type:** void*()<br>**Scope:** main, class Teacher |
| | Control Jujutsu NGINX [47] | **Corrupted:** ctx->output_filter<br>**Target:** ngx_execute_proc() | **Type:** ngx_output_chain_filter_pt<br>**Scope:** ngx_output_chain | **Type:** static void *(ngx_cycle_t *cycle, void* data)<br>**Scope:** ngx_execute |
| | CVE-2014-8668 - libtiff v4.0.6 | **Corrupted:** tif->tif_encoderow<br>**Target:** Arbitrary pointer | **Type:** TIFFCodeMethod<br>**Scope:** _TIFFSetDefaultCompression, TIFFWriteScanline, TIFFOpen, main | Unknown, since this is a CVE and not an attack |
| | CVE-2014-1912 - python-2.7.6 | **Corrupted:** tp->tp_hash<br>**Target:** Arbitrary pointer | **Type:** hashfunc<br>**Scope:** inherit_slots, PyObject_Hash | Unknown, since this is a CVE and not an attack |
| **Data oriented attack** | DOP ProFTPd Attack [57] | **Corrupted:** &ServerName<br>**Target:** resp_buf, ssl_ctx | **Type:** const char*<br>**Scope:** core_display_file | **Type:** char*<br>**Scope:** pr_response_send_raw |
| | NEWTON CPI Attack [113] | **Corrupted:** v[index].get_handler<br>**Target:** dlopen | **Type:** ngx_http_get_variable_pt<br>**Scope:** ngx_http_get_indexed_variable | **Type:** void* (const char *filename, int flags)<br>**Scope:** ngx_load_module |

### 3.4.8 Merging of Compatible Types for Casting

Depending on which mechanism is used and the pointer casts, different RSTI types can be combined or merged together. This is showcased in Figure 3.8. There are two basic types, void* and int*, but each RSTI mechanism distinguishes between them depending on the code. RSTI-STC merges types if there is a cast (Line 5), and thus both basic types would have one RSTI-type under RSTI-STC. RSTI-STWC doesn't merge types with casts. However, since p1 and p2 have the same scope, type, and permission, they both will have one RSTI-type. RSTI-STL adds the location, and this allows it to distinguish p1, p2, and p3 with three RSTI-types. This distinction directly affects the number of instrumentations for each mechanism.

## 3.5 Implementation

Our prototype is built on top of Apple's LLVM fork [1]. It consists of 3504 lines of code (LoC), with 3017 LoC for the LLVM pass and 487 LoC for the pointer-to-pointer library. This includes all three mechanisms. The pointer-to-pointer library is integrated into LLVM's compiler-rt. The LLVM pass executes on the IR level but is in the AArch64 backend.

We apply a few optimizations to RSTI. First, is Link Time Optimization (LTO), which combines all the object files into one file. Also, we inline all the pointer-to-pointer library functions. In addition to that, we execute the pass in the LTO phase. This allows RSTI to avoid unnecessary instrumentation.

Executing the pass in the LTO phase, particularly after all the object files have been combined into one, is not only beneficial for performance reasons, but is important to help RSTI function properly and efficiently. Being able to have a full look at the program and analyze all the scope-type information at once, instead of doing it per object file, allows STI to completely map all the scope-type information for all the pointers accurately for the entire program.

## 3.6 Evaluation

We evaluate RSTI by answering the following questions:

- How effective is RSTI in preventing state-of-the-art attacks as well as real-world attacks? (§ 3.6.1)
- How secure is RSTI against abusing pointer-to-pointer metadata and pointers with the same RSTI-type? (§ 3.6.2)
- How much performance overhead does RSTI impose in benchmarks and real-world pro-

**Table 3.2:** Security evaluation summary. This table shows how each RSTI mechanism constricts an attacker, as well as their security guarantees.

| Technique | RSTI-STC | RSTI-STWC | RSTI-STL |
|---|---|---|---|
| **How it works** | Constricting variables to their scope, type and permission, and handle casts by combining into one compatible type | Constricting variables to their scope, type and permission, and handle casts by authenticating/re-signing a pointer | Constricting variables to their scope, type, permission and location without combining. In other words, adding &p to the modifier. |
| **Attacker restriction** | The attacker here can substitute pointers if they both have a valid PAC and as well as have the same RSTI-type. | Similar to RSTI-STC, the attacker can substitute pointers if they both have a valid PAC, as well as have the same RSTI-type. | The attacker here cannot even substitute a pointer at a separate location. |
| **Defense capability** | **Pointer corruption:** An attacker cannot substitute with pointers that have different RSTI-types. However, the size of the RSTI-type may be large due to combining. **Spatial Safety:** An attacker overflowing the buffer needs to override with a pointer or location that is of the same RSTI-type. **Temporal Safety:** Similar to spatial safety, an attacker needs to reuse the freed pointer within the same scope and type. | **Pointer corruption:** An attacker cannot substitute with pointers that have different RSTI-types. Due to not combining, RSTI-STWC is stronger than RSTI-STC. **Spatial Safety:** Achieving an attack through spatial violations is harder, due to stricter RSTI-type at that pointer. **Temporal Safety:** Similar to spatial safety, an attacker needs to reuse the freed pointer with the same RSTI-type, but stronger than RSTI-STC. | **Pointer corruption:** Here, the pointer would have no substitutes, meaning that only that pointer can be dereferenced from that location. **Spatial Safety:** It wouldn't be possible to abuse a pointer with a spatial safety violation. A buffer overflow would always be detected due to &p. **Temporal Safety:** Similar argument to spatial safety, any usage of the freed pointer with any other pointer or at another location would be detected due to &p. |

grams? (Section 3.6.3)

### 3.6.1 Security Evaluation

This section evaluates RSTI's effectiveness in stopping security attacks. We first explain the different types of attacks (Section 3.6.1), and how RSTI defends against these attacks (Section 3.6.1).

#### Attacks Landscape

In this section, we show that RSTI can defend against both control-flow hijacking and data-oriented attacks, as well as real-world exploits. Table 3.1 shows a variety of state-of-the-art attacks. We exercised RSTI against powerful synthesized attacks, such as NEWTON [113] and AOCR [96], as well as C++-specific attacks, such as COOP. We also chose real-world attacks from exploits in libtiff and Python. Lastly, we evaluate RSTI against a Data Oriented Programming (DOP) attack [57] that leaks an SSL key.

### Attacks in Detail and How RSTI Defends

As Table 3.1 indicates, in all the known attacks, pointers are abused and the change in scope-type information allows the attack to be detected by RSTI. We only explain two of these attacks in detail due to space limitations.

**NEWTON CsCFI attack.** This is an attack on NGINX that calls and exploits mprotect. One of the attack steps is maliciously overwriting a function pointer (c->send_chain) in the function ngx_http_write_filter with a pointer to malloc. If the malloc pointer is in libc, then it won't have a PAC on it, and thus authentication would fail. Moreover, if we assume that both the function pointer and the malloc pointer are protected by RSTI, then authentication would still fail due to both having different types (void*(size_t size) and ngx_send_chain_pt), as well as being in different scopes. Thus, RSTI detects this attack.

**DOP ProFTPd attack.** This is a Data Oriented Programming attack [57] which corrupts the first known pointer of struct ssl_ctx in a loop and overwrites it with 8 malicious dereferences, relying on 4 gadgets for each dereference. Since these dereferences invoke load gadgets, these gadgets are protected by RSTI, and thus cannot be invoked maliciously without conforming to the RSTI-type. This load gadget corrupts &ServerName with data from resp_buf. However, &ServerName and resp_buf have different RSTI-types. &ServerName is of type const char* and its scope is core_display_file, while resp_buf is of type char* with scope pr_response_send_raw. Since RSTI protects data pointers with RSTI-type, RSTI detects this attack. This attack can only succeed if all dereferences and gadgets have the same RSTI-type.

Table 3.2 summarizes the RSTI techniques, attacker restrictions, and defense capabilities of each technique against pointer corruption, spatial safety and temporal safety.

**Comparison against prior works.** RSTI offers a refined type compared to other

**Table 3.3:** SPEC 2006 equivalence class data. (NT: Number of types in the program; RT: Number of RSTI-types; NV: Total number of pointer variables; $EC_V$: Equivalence class of variable; $EC_T$: Equivalence class of type. )

| BM | NT | RT | | NV | Largest $EC_V$ | | Largest $EC_T$ | |
|---|---|---|---|---|---|---|---|---|
| | | STC | STWC | | STC | STWC | STC | STWC |
| perlbench | 155 | 318 | 722 | 2939 | 198 | 82 | 33 | 1 |
| bzip2 | 25 | 31 | 55 | 122 | 32 | 13 | 7 | 1 |
| mcf | 12 | 35 | 40 | 95 | 9 | 8 | 2 | 1 |
| milc | 55 | 154 | 195 | 440 | 54 | 18 | 18 | 1 |
| namd | 30 | 73 | 100 | 230 | 23 | 23 | 10 | 1 |
| gobmk | 120 | 216 | 417 | 1057 | 111 | 46 | 25 | 1 |
| dealII | 2546 | 4528 | 8878 | 21018 | 676 | 44 | 192 | 1 |
| soplex | 129 | 970 | 1690 | 3399 | 137 | 27 | 66 | 1 |
| povray | 282 | 620 | 1446 | 3791 | 229 | 25 | 76 | 1 |
| hmmer | 90 | 198 | 405 | 973 | 56 | 24 | 16 | 1 |
| libquantum | 13 | 33 | 44 | 58 | 9 | 4 | 5 | 1 |
| sjeng | 29 | 47 | 73 | 130 | 19 | 9 | 7 | 1 |
| h264ref | 116 | 252 | 354 | 727 | 48 | 23 | 15 | 1 |
| lbm | 14 | 14 | 20 | 33 | 12 | 7 | 4 | 1 |
| omnetpp | 255 | 558 | 1241 | 2458 | 94 | 26 | 31 | 1 |
| astar | 36 | 59 | 98 | 156 | 18 | 11 | 12 | 1 |
| sphinx3 | 88 | 188 | 321 | 686 | 36 | 20 | 12 | 1 |
| xalancbmk | 2558 | 7503 | 14073 | 32097 | 603 | 122 | 206 | 1 |

prior data pointer integrity works that use PAC, such as PARTS [74], and thus has better coverage against pointer substitution attacks. For example, in the DOP ProFTPD attack in Table 3.1, the corrupted and original pointers are both of type char*. PARTS wouldn't be able to detect this, since it only relies on type. However, RSTI's refined scope-type detects it. The PittyPat attack is another example. Thus, RSTI can mitigate more attacks.

### 3.6.2 Analysis on RSTI Instrumentation

#### Equivalence Class

Now we want to showcase the usefulness of the RSTI-type information. We define some terminology first:

- **Number of types in program (NT)**: This is the number of basic types a program has, such as int*, void*, etc.
- **Number of RSTI-types (RT)**: This refers to the actual types that are enforced by the specific RSTI mechanism.
- **Equivalence Class of Type ($\mathbf{EC_T}$)**: RSTI-STWC has the advantage of having only *one basic type* for each *RSTI-type*, whereas RSTI-STC can have more than one basic type in an RSTI-type, due to the combining. We refer to the number of basic types in each RSTI-type as *Equivalence Class of Type ($EC_T$)*. We use the term *equivalence class* since this count provides a measure of how viable pointer substitution attacks can be within an application. The same term is used in CFI papers and RSTI is the first, as far as we know, to adopt it for data pointers.
- **Equivalence Class of Variable ($\mathbf{EC_V}$)**: RSTI-STWC has a maximum $EC_T$ of 1, due to there being only one type in each RSTI-type. However, RSTI-STWC does not have *one variable* for each RSTI-type. Several variables can be declared within the same RSTI-type. We refer to the number of variables in each RSTI-type as *Equivalence Class of Variable ($EC_V$)*.The largest $EC_V$ for RSTI-STL is always 1, due to the inclusion of the location (&p) in the modifier, while the largest $EC_V$ for RSTI-STWC would vary.

Table 3.3 shows the EC data for SPEC CPU2006. RSTI increases the number of types that can be distinguished in a program. Also, the impact of combining RSTI-types for RSTI-STC reduces the number of RSTI-types in the mechanism, but is still higher than

the number of types without RSTI. RSTI-STL is not shown in the table. This is because the largest $EC_T$ and $EC_V$ for RSTI-STL are always 1, due to the enforcement of the location. RSTI-STL is the most secure, however RSTI-STWC does still provide security value due to its $EC_T$ being always 1, and significantly reducing $EC_V$.

#### Pointer-to-pointer Data from SPEC 2006

We had previously explained our exact use case of pointer-to-pointer handling in Section 3.4.7. Our intuition was that this exact case of losing the original type of the pointer when a pointer-to-pointer is casted and passed as an argument to a function is rare. And this was confirmed by our analysis of SPEC 2006. There is a total of 7489 sites across the benchmarks where a pointer-to-pointer is passed or loaded. Of those, only 25 meet the special criteria where the original type could be lost. This confirms our intuition that this is a rare case.

### 3.6.3 Performance Evaluation

#### Methodology

**Evaluation environment.** Our evaluations were run on the Apple M1 [11], which supports the ARMv8.4 architecture and ARM PA. We used an Apple Mac Mini M1 [12], that has 4 small cores, 4 big cores, and 8GB DRAM. Our prototype was implemented on Apple's LLVM fork [1]. We compiled all the applications with LTO and O2 for fair comparison.

**Evaluating C programs.** All our C programs were run with real PA instructions. We were able to disable Apple's use of PA [13] to avoid any conflicts that would happen between RSTI's and Apple's PA instrumentation.

**Evaluating C++ programs.** Whilst evaluating C++ programs, we discovered that

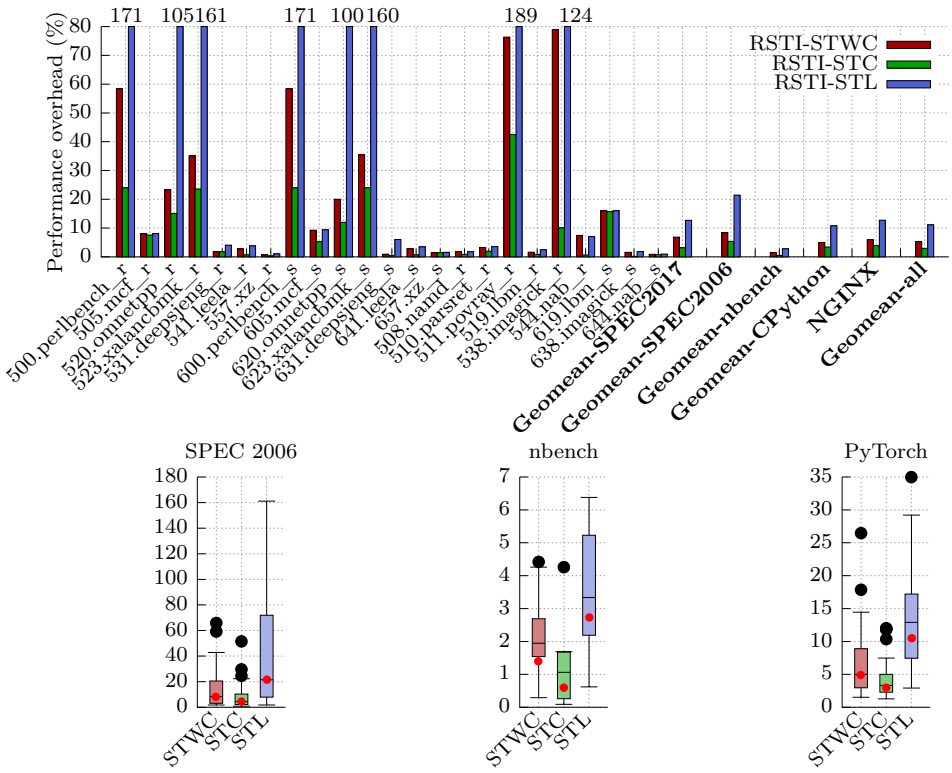



**Figure 3.9:** The performance overhead of SPEC CPU2017, and the geometric means of SPEC CPU2006, nbench, CPython Pytorch and NGINX for all three RSTI mechanisms. The box plot shows the minimum, median, maximum and quartile values. The black dots represent outlier values. The red dots represent the geometric mean.

PA instructions are built into Apple's standard C++ library. Thus, we were not able to turn them off. So, for C++ applications, we first did a correctness test on ARM's Fixed Virtual Platform (FVP) [14]. Then, in order to emulate the performance overhead of the PA instructions, we used seven XOR (eor) instructions as an equivalent to one PA instruction on the Mac Mini M1. This has been measured and confirmed in previous works [60, 72, 117].

**Benchmarks.** For our performance evaluation, we used a wide variety of benchmarks to showcase RSTI's versatility, namely: SPEC CPU 2006 [55], SPEC CPU 2017 [22], and nbench [82]. We ported the SPEC CPU 2006 benchmarks to the Apple M1 and built them from scratch. We were not able to run 403.gcc and 625.x264 on the M1, in spite of using

Apple's own Clang compiler. We think that there is a bug in the macOS toolchain version that we used.

### Performance Overhead

Figure 3.9 shows the performance overhead for the SPEC CPU2017 benchmarks, as well as NGINX and the geometric mean of SPEC CPU2006, nbench and CPython PyTorch. The individual performance numbers for SPEC CPU2006, nbench and CPython PyTorch were not included due to space limit. Thus, we use boxplots to show the distribution.

**SPEC CPU2017.** SPEC 2017 benchmarks have a geometric mean of 6.86%, 3.17%, and 12.70% for RSTI-STWC, RSTI-STC, and RSTI-STL respectively. Some benchmarks, such as perlbench, povray, and xalancbmk have exceptionally higher overhead. These benchmarks are known to heavily dereference pointers, either in a loop or very frequently [60].

**SPEC CPU2006.** SPEC 2006 benchmarks have a geometric mean of 8.42%, 5.36%, and 21.47% for RSTI-STWC, RSTI-STC, and RSTI-STL, respectively. We analyzed the instrumentation and found the performance overhead is highly correlated with the number of instrumented load/stores, with a Pearson correlation factor of 0.75-0.8. There are some exceptions, due to loops or some of the instrumented loads/stores never getting called, but the overall results are consistent.

**CPython 3.9.** For CPython3.9, we evaluated PyTorch benchmarks [6] to test how RSTI would perform if used in a machine learning context.The CPython PyTorch benchmarks have a geometric mean of 5.01%, 3.44%, and 10.80% for RSTI-STWC, RSTI-STC, and RSTI-STL, respectively. Thus, RSTI can be utilized in machine learning scenarios where additional security guarantees are needed.

**NGINX.** We also evaluated NGINX, a real-world application with RSTI on the Apple M1. We stress its 4 big cores and use the same configuration used to test NGINX TLS transactions per second [83]. We relied on wrk [50], an HTTP benchmarking tool, to stress test NGINX by generating concurrent HTTP requests. wrk was run on a separate machine on the same network and spawned three threads, with each thread handling 50 connections. The overhead was 5.98%, 3.93%, and 12.76% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

The total geometric mean across all the benchmarks and applications is 5.29%, 2.97% and 11.12% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

**Overall.** As can be seen, the RSTI mechanisms generally have mild overhead. Some exceptions are there for RSTI-STL, where performance overhead can exceed more than 100%. This is still comparatively lower to other defense mechanisms such as DFI, which has an average overhead of around 106%, and YARRA, which has an average overhead of 400%-600% when protecting the whole program. However, real programs, such as NGINX and PyTorch, have a reasonable overhead. Thus, the RSTI mechanisms can be effectively used to secure machine learning and real-life applications.

**Comparison against prior works** RSTI offers lower overhead compared to other PAC-based data pointer defense mechanisms. One of these is PARTS [74]. PARTS is evaluated only on the nbench benchmarks, and the mean overhead is 19.5%. In comparison, RSTI's mean overhead on nbench is 1.54%, 0.52% and 2.78% for RSTI-STWC. RSTI-STC and RSTI-STL respectively. Although both PARTS and RSTI instrument data pointers, our design choices, such as using LLVM ptrauth intrinsics, running the pass in the backend, using LTO and -O2 optimizations allowed our compiler to produce much more optimized code. From our correspondence with the PARTS authors, they ran their evaluation with -O0. Our design choices allow us to have lower performance overhead than PARTS whilst

offering higher security guarantees.

## 3.7 Discussion

**Comparison with memory safety.** Due to the constricting nature of the RSTI mechanisms and due to the fact that RSTI protects all pointers in a program, RSTI provides a static alternative to traditional memory safety techniques for providing protection from spatial and temporal safety. Bear in mind that RSTI does not prevent spatial and temporal memory errors but prevents an attacker from abusing them. In addition to that, RSTI does not instrument non-pointer variables, and they are out of scope of this work. The design can be extended to include offsets and indexes by converting them to pointers to cover them, but we relegate this to future work.

**Metadata attack in RSTI.** Pointer-to-pointer metadata is the only metadata that is stored in memory. However, the attacker cannot see the actual types in memory. In our implementation, each type is assigned a type ID in the internal LLVM data structure during compilation. When storing pointer-to-pointer metadata, that type ID is used to represent each type. Thus, the metadata information is not meaningful to the attacker. In addition to that, the metadata is read-only and can only be accessed by the RSTI pointer-to-pointer library.

**Matching scope-type information.** There are cases where pointers could have the same RSTI-type. Since RSTI needs to maintain program semantics, distinguishing between pointers that have the same RSTI-type is hard statically. However, it is not practically feasible for an attacker to maintain the same RSTI-type when executing an attack, as can be seen in § 3.6.1.

**Is the PAC length enough?** In macOS, and in our setting, the PAC length is 16 bits, with the exception of pointer-to-pointer where the PAC length is 8 bits. This means that, for 16 bits, there are 65,536 possible PACs, which is sufficient for most practical cases. Prior work [72] has shown that real-life applications, such as NGINX and Apache usually have around 25,000 live objects on average. Thus, there is more than enough PAC combinations. Since the PAC length on macOS is 16 bits, the probability for a PAC collision when targeting a single object is $1/2^{16}$ which is equal to $1.52 \times 10^{-5}$. This is considered to be negligible, and does not give an attacker practicality to commit an attack [56]. As for the 8 bits in the case of pointer-to-pointer, this allows for 256 possible PACs, and we showed in Section 3.6.2 that there are only 25 cases in the entirety of SPEC 2006 that needed this special handling.

**Protecting return addresses.** The use of scope-type information to protect return addresses is not as powerful as using it to protect pointers. There are many practical and strong defenses that protect return addresses, and this is considered orthogonal RSTI. RSTI can be deployed alongside other return address protections. For example, in our evaluations, we turn on the default Apple return address protection scheme.

## 3.8 Related Work

**Type-based defenses.** The goal of EffectiveSan [43] is to do bounds checking by combining type checking with low fat pointers. TDI [85] relies on grouping types into specific memory arenas by relying on a special allocator and compiler instrumentation. Type after type [110] replaces regular allocations with typed allocations that never reuse memory.

**ARM PAC defenses.** PARTS [74] protects pointers with pointer authentication by relying on the LLVM ElementType as the modifier. AOS [65] utilizes the PAC as an index

to access a bounds table and do bounds checking.

**Data oriented defenses.** Data Flow Integrity (DFI) [29] makes sure that a variable can only be written by its legitimate write instruction. Hardware assisted Data Flow Isolation (HDFI) [104] provides instruction-level isolation by relying on tags. YARRA [98] is an extension of the C language that relies on programmer annotations to protect critical data types.

**Control-flow hijacking defenses.** Control-Flow Integrity (CFI) [8] constricts the number of valid target sites for an indirect control-flow transfer. Code Pointer Integrity (CPI) [69] protects a subset of pointers, referred to as sensitive pointers.

## 3.9 Conclusion

This paper introduced Scope-Type Integrity (STI), a new defense policy that enforces a pointer to conform to the programmer's intended usage by utilizing scope, type and permission information and bringing that information back to leverage during runtime. We presented RSTI, a robust and efficient security mechanism that protects all pointers in a program by leveraging ARM Pointer Authentication. RSTI enforces STI to ensure that each pointer conforms to its scope, type, and permission that was originally intended by the programmer. RSTI brings back this information to the runtime. We implemented three RSTI defense mechanisms with varying levels of security granularity, enforcing control-flow and data-flow integrity. We showcased the security of RSTI against state-of-the-art synthesized and real attacks, and demonstrated its low performance overhead across a variety of benchmarks and and real-life applications. We plan to open source RSTI upon paper acceptance.

# Chapter 4

# Ongoing Research and Future project

This chapter of the thesis proposal outlines the proposed post-preliminary exam future work. For the future work, this proposal proposes extending protection with hardware security features to cover speculative execution memory corruption attacks. The intention is to have a new design that utilizes Pointer Authentication and other hardware security features, if needed, to design a defense mechanism capable of defending against specific variants of Spectre [67] without needing to extend the hardware. Section 4.1 gives a brief background on Spectre and speculative execution memory corruption attacks. Section 4.2 gives a brief idea on what the proposed defense would look like and the rationale behind a possible design.

## 4.1 Speculative Execution memory corruption attacks

As mentioned before, memory corruption vulnerabilities have existed for a long time and have been a plague to computer systems and a lucrative avenue for attackers looking to compromise systems. Many mechanisms have been proposed and deployed over the years to mitigate them, as discussed before in earlier sections of this proposal.

Recently, a new class of attacks has emerged, referred to as transient execution attacks [25, 116]. In our case, we are focused more specifically on speculative execution attacks [19, 66, 67, 68, 78, 79], and these are attacks that cause the program to speculatively perform certain operations that would not be executed if the program was executing correctly. The results of this execution would then be leaked via side channels, such as the cache, that the attacker can observe.

The main issue with PACTight and RSTI is that they rely on the effects of the attack actually being visible in memory, and do not take account for speculative execution. PACTight relies on an external metadata that gets updated only when instructions are actually executed, and RSTI doesn't have a dynamic metadata, and thus the attacker can correctly execute the instructions and leak data speculatively if they want. This means that a new mechanism is needed to combat these attacks.

## 4.2 Pointer Authentication based Spectre defense

This thesis proposal still proposes to use Pointer Authentication for defending, but to combine that with lfences. lfences are instructions that prevent speculative exection from happening until the lfence instruction has been executed. In order for the attacker to execute a speculative execution attack, the attacker needs a speculative gadget, which would usually be a load instruction of a pointer to the sensitive data that exists inside an if statement. This proposal proposes instrumenting all pointers with pointer authentication instructions, similar to RSTI. However, the load instruction to the sensitive data would be isolated via programmer annotation and given its own unique modifier and an lfence instruction added to it. In this way, the attacker cannot abuse that instruction, and if the attacker tries to use another load instruction, that load instruction is protected with pointer authentication.

There is still an ongoing discussion on the viability of this design approach.

Another avenue of that is being investigated is possible defenses against speculative execution attacks in Rust. SPEAR [80] proposes a speculative execution attack against Rust that speculatively violates Rust's bound checking mechanism. This is important, since one of Rust's main features is its memory safety and prevention of memory corruption attacks that have plagued C/C++. Preventing this type of attack would be an important contribution.

# Chapter 5

# Conclusions

This thesis presents research projects that utilize hardware security features to design practical and efficient defense mechanisms against control-flow hijacking, data-oriented and speculative execution attacks.

This proposal discussed prior work, 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, against many benchmarks including SPEC 2006, nbench and CoreMark using real PAC instructions.

This proposal introduced Scope-Type Integrity (STI), a new defense policy that enforces a pointer to conform to the programmer's intended usage by utilizing scope, type and permission information and bringing that information back to leverage during runtime. We presented RSTI, a robust and efficient security mechanism that protects all pointers in a program by leveraging ARM Pointer Authentication. RSTI enforces STI to ensure that each pointer conforms to its scope, type, and permission that was originally intended by the programmer. RSTI brings back this information to the runtime. We implemented three RSTI defense mechanisms with varying levels of security granularity, enforcing control-flow and data-flow integrity. We showcased the security of RSTI against state-of-the-art synthesized and real attacks, and demonstrated its low performance overhead across a variety of benchmarks and and real-life applications.

Finally, this proposal introduced a new direction for utilizing hardware security features to protect programs from speculative execution attacks.

# Bibliography


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

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

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

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

[5] Llvm pointer authentication. https://llvm.org/docs/PointerAuth.html.

[6] PyTorch Benchmarks. https://github.com/pytorch/benchmark.

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

[15] Arm. Top Byte Ignore, 2018. https://en.wikichip.org/wiki/arm/tbi.

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

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

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

[19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479.

[20] 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.

[21] 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.

[22] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356299.

[23] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[24] 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.

[25] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[26] 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.

[27] 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.

[28] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 193–204, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444.

[29] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, Seattle, WA, November 2006.

[30] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[31] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[32] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[33] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Vik: practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2022.

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

[35] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[36] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection Against Function-reuse Attacks. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[37] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[38] 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.

[39] 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.

[40] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[41] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 131–148, Vancouver, BC, Canada, August 2017.

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

[43] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985.

[44] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132.

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

[46] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Miss-

ing the point (er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] Jens Grossklags and Claudia Eckert. $\tau$CFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Heraklion, Crete, Greece, September 2018.

[53] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Pro-*

*ceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[54] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325.

[55] 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.

[56] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Pacsafe: Leveraging arm pointer authentication for memory safety in c/c++, 2022.

[57] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[58] 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.

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

[60] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with PACTight. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3717–3734, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1.

[61] Ismail, Mohannad and Yom, Jinwoo and Jelesnianski, Christopher and Jang, Yeongjin and Min, Changwoo. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1612–1626, 2021.

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

[63] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, 2019.

[64] 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.

[65] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1153–1166, 2020.

[66] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018. URL http://arxiv.org/abs/1807.03757.

[67] 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.

[68] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association. URL https://www.usenix.org/conference/woot18/presentation/koruyeh.

[69] 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.

[70] 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.

[71] 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.

[72] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1901–1915, New York, NY, USA, 2022.

Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560598.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer*

*and Communications Security*, CCS '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930.

[79] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association. URL https://www.usenix.org/conference/woot19/presentation/mambretti.

[80] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *IEEE European Symposium on Security and Privacy*, pages 633–649, 2021. doi: 10.1109/EuroSP51992.2021.00048.

[81] 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.

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

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

[84] 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.

[85] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1049–1065, 2022.

[86] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[87] 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.

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

[89] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[90] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

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

[92] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[93] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[94] 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.

[95] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 685–698, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104.

[96] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[97] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 131–145, 2011.

[98] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[99] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Diffi-

culty 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.

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

[101] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Cr-count: 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.

[102] 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.

[103] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[104] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Pack. HDFI: Hardware-Assisted Data-flow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[105] Nigel Stephens. Developments in the Arm A-Profile Architecture: Armv8.6-A, 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a.

[106] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory.

In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[107] The Clang Team. Clang 10 documentation: CONTROL FLOW INTEGRITY, 2019. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[108] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[109] 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.

[110] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 17–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697.

[111] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[112] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings*

*of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[113] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[114] 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.

[115] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[116] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 2021.

[117] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1.

[118] 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.

[119] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[120] 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.

[121] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[122] 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.