Practical Mitigations Against Memory Corruption and Transient Execution Attacks

Mohannad A. Ismail

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Wenjie Xiong, Co-chair
Changwoo Min, Co-chair
Danfeng (Daphne) Yao
Haining Wang
Michael Hsiao

April 25, 2024
Blacksburg, Virginia

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

Practical Mitigations Against Memory Corruption and Transient Execution Attacks

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. In addition to that, a new class of attacks, Spectre, has emerged that abuse speculative execution to leak secrets and sensitive data through microarchitectural side channels. Many defenses have been proposed to mitigate Spectre as well. However, the current mitigations against memory corruption and Spectre attacks still have many limitations.

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 dissertation describes utilizing hardware security features, namely ARM PAC, to build effective and practical defense mechanisms. This dissertation 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 dissertation 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. This dissertation presents 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.

Also, this dissertation proposes sPACtre, a new and novel defense mechanism that aims to prevent Spectre control-flow attacks on existing hardware. sPACtre is an ARM-based defense mechanism that prevents Spectre control-flow attacks by relying on ARM's Pointer Authentication hardware security feature, annotations added to the program on the secrets that need to be protected from leakage and a dynamic tag-based bounds checking mechanism for arrays. This dissertation shows that sPACtre can defend against these attacks. This dissertation evaluates sPACtre on a variety of cryptographic libraries with several cryptographic algorithms, as well as a synthetic benchmark, and show that it is efficient and has low performance overhead

Finally, this dissertation explains a new direction for utilizing hardware security features to protect energy harvesting devices from checkpoint-recovery errors and malicious attackers.

Practical Mitigations Against Memory Corruption and Transient Execution Attacks

Mohannad A. Ismail

(GENERAL AUDIENCE ABSTRACT)

In recent years, cyber-threats against computer systems have become more and more prevalent. In spite of many recent advancements in defenses, these attacks are becoming more threatening. However, many of these defenses are not implemented in the real-world. This is due to their high performance overhead. This limited efficiency is not acceptable in the real-world. In addition to that, many of these defenses have limited coverage and do not cover a wide variety of attacks. This makes the performance tradeoff even less convincing. Thus, there is a need for effective and practical defenses that can cover a wide variety of attacks.

This dissertation first provides a comprehensive overview of the current state-of-the-art and most dangerous attacks. More specifically, three types of attacks are examined. First, control-flow hijacking attacks, which are attacks that divert the proper execution of a program to a malicious execution. Second, data oriented attacks. These are attacks that leak sensitive data in a program. Third, Spectre attacks, which are attacks that rely on supposedly hidden processor features to leak sensitive data. These ”hidden” features are not entirely hidden. This dissertation explains these attacks in detail and the corresponding state-of-the-art defenses that have been proposed by the security research community to mitigate them.

This dissertation then discusses effective and practical defense mechanisms that can mitigate these attacks. The dissertation discusses past work, PACTight, as well as its contributions,

RSTI and sPACtre, presenting the full design, threat model, implementation, security evaluation and performance evaluation of each one of these mechanisms. The dissertation relies on insights derived from the nature of the attack and compiler techniques. A compiler is a tool that transforms human-written code into machine code that is understandable by the computer. The compiler can be modified and used to make programs more secure with compiler techniques. The past work, PACTight, is a defense mechanism that defends against the first type of attacks, control-flow hijacking attacks, by preventing an attacker from abusing specific code in the program to divert the program to a malicious execution. Then, this dissertation presents RSTI, a new defense mechanism that overcomes the limitations of PACTight and extends it to cover data oriented attacks and prevent attackers from leaking sensitive data from the program. In addition to that, this dissertation presents sPACtre, a novel defesnse mechanism that defends against Spectre attacks, and prevents an attacker from abusing a processor's hidden features. Finally, this dissertation briefly discusses a possible future direction to protect a different class of devices, referred to as energy-harvesting devices, from attackers.

To my Mom and Dad.

For their endless sacrfice and unwavering love.

Acknowledgments

In the name of Allah the Most Gracious and the Most Merciful.

Writing and finishing this dissertation would not have been possible if it was not for all the amazing and wonderful people who helped me during my PhD journey. It has been an unforgetable and life-changing experience.

I would like to start off by thanking my advisors, Dr. Wenjie Xiong and Dr. Changwoo Min. They are both great mentors and advisers. Their guidance, help and patience have been invaluable to me and I wouldn't be where I am without their continuous support. Thank you Dr. Xiong for agreeing to join in as my adviser and help me finish my degree. Working with you this past year has been nothing short of amazing. I enjoyed our discussions and diving into research topics during meetings. I feel lucky and fortunate to have had you as an adviser. Thank you Dr. Min for your unconditioned support and guidance the past few years. You have given me an opportunity to achieve my dreams and I will always be grateful for that. You have always been there for me and I never hesitated to ask you any questions.

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

A special thank you to Dr. Yeongjin Jang, Andrew Quach and Soham Gandhi for their valuable contribution to this work. Their help has been invaluable and their expertise contributed to accomplishing this work.

I am also thankful to my colleagues in the previously known Computer Systems, Memory and OS Security (COSMOSS) Lab. We are all in different paths now, but I am thankful to be able to know you. Your contributions to my research helped in completing my dissertation.

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

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, attacks against computer systems have become more sophisticated and more prevalent. These attacks can be either software-based attacks, such as control-flow hijacking and data-oriented attacks, or they can be hardware-based attacks, such as speculative execution attacks.

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) [24] 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 [93] and NEWTON [171] 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 [24, 49, 59, 70,

81, 85, 86, 94, 100, 106, 118, 135, 136, 140, 142, 164, 165, 169, 170, 184, 186]. 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.

Spectre attacks. Speculative execution is a micro-architectural optimization technique in which a processor executes certain instructions in advance anticipating that their results may be needed in the near future. Speculative execution has many performance benefits. If the speculation path does get taken, the processor doesn't have to execute these instructions, since they were already executed and the processor can use the results. If the speculation path does not get taken, these results are squashed architecturally. However, this is where a security issue arises. Even though the results of the speculative execution are no longer available architecturally, they are still available in the micro-architecture, such as in the cache. This phenomena has given rise to Spectre attacks [103], a class of attacks that abuse speculative execution to leak possible secrets through micro-architectural side channels. Spectre attacks are powerful, since they cannot be easily defended against without severely impacting the performance, and negating the benefits of speculative execution. There have been many Spectre attacks proposed [43, 51, 90, 92, 102, 103, 105, 114, 121, 144, 147, 176, 178], exploiting many different micro-architectural effects and leakages that result from Spectre. This has resulted in many *Spectre variants*. A specific subset of these variants are referred to as *Spectre control-flow attacks*. These are Spectre attacks that rely on abusing the control-flow of the program speculatively to leak secrets to the side channel.

Challenges in Current State-of-the-art mitigations. There have been many defenses that have been proposed to defend against these attacks. However, they suffer from practicality issues. Defending against this wide variety of attacks is difficult, especially when there is a need for the defense to be practical.

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 and transient execution attacks.

- **Low coverage** Some current defense mechanisms such as PTAuth [78], PACStack [112], most CFI defense mechanisms [24, 49, 59, 70, 81, 85, 86, 94, 100, 118, 135, 136, 140, 142, 164, 165, 169, 170, 184, 186] and Spectre defenses [3, 10, 11, 12, 21, 22, 60, 62, 80, 102, 133, 157, 172, 174, 183, 188] limit themselves in terms of coverage. PACStack only protects return addresses, and PTAuth only provides temporal memory safety. On the Spectre defenses side, Blade [172] and oo7 [174] only defend against one variant of Spectre. 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) [106] 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 [76], 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 [132] and DangSan [167] have average overheads that exceed 100%. Venkman [157] and Intel-lfence [12] have overheads up to 300% when defending against Spectre attacks. A defense mechanism needs to be practical in order to provide justification for its use

in real-world applications.

- **Compatibility and deployability** 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. In addition to that, they need to be deployable on current existing hardware. Speculative Taint Tracking (STT) [183] selects and forwards specific results of speculative execution if it can prove that the forwarded results do not transmit to a covert side channel. Speculative Privacy Tracking (SPT) [60] delays the execution of a transmitter until it can prove that its operands leak during non-speculative execution, and thus it is safe. Both STT and SPT have simulator implementations and thus aren't deployable on existing hardware.

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 dissertation aims to address the following questions:

- Is it possible to design an effective and practical defense mechanism to defend against memory corruption attacks and transient execution 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.

Hardware vendors such as Intel and ARM have proposed hardware security features that aim to help in mitigating these attacks. Some of these hardware security features can be utilized to develop practical defenses in a hardware-software co-design manner.

This dissertation 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 dissertation 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 first current work is Scope Type Integrity (STI), 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. The second current work is sPACtre, a new defense mechanism that aims to thwart Spectre control-flow attacks, and these are Spectre-PHT, which abuses branch prediction, Spectre-BTB, which abuses the branch target predictor, and Spectre-RSB which abuses the return stack buffer.

Contributions. The main goal of this dissertation is to propose effective and practical defense mechanisms using hardware security features. Specifically, this dissertation 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 dissertation aims to design solutions that covers a wide array of memory corruption attacks, specifically a wide variety of pointers. Specifically, this disseratation 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 dissertation 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 dissertation is lower than their counterparts in the state-of-the-art.
- **Compatibility and deployability** 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. In addition to that, the proposed defense mechanism are readily deployable on currently existing hardware.

This dissertation 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 dissertation 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 dissertation proposes sPACtre, a new and novel defense mechanism that aims to prevent Spectre control-flow attacks on ARM by relying on a combination of software defense mechanisms and existing hardware security features to provide an efficient and effective defense mechanism. In this way, these defense mechanisms defend against a wide variety of system security attacks.

Therefore, this dissertation makes the following contributions:

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

- The design, implementation and evaluation of sPACtre, a new and novel defense mechanism that aims to prevent Spectre control-flow attacks on existing hardware. sPACtre is an ARM-based defense mechanism that prevents Spectre-PHT, Spectre-BTB and Spectre-RSB attacks by relying on ARM's Pointer Authentication hardware security feature, annotations added to the program on the secrets that need to be protected from leakage and a dynamic tag-based bounds checking mechanism for arrays.

Dissertation Organization. This dissertation is organized as follows. Chapter 2 presents background information regarding the dissertation. This includes more detailed information on the relevant attacks, as well as the hardware defense mechanism, ARM Pointer Authentication. Section 2.5 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 presents sPACtre, a defense mechanism that aims to defend against Spectre control-flow attacks. Chapter 5 shows a possible future direction of research in which hardware security features can be utilized to protect energy harvesting systems. These are are performance-critical systems that are currently vulnerable to a variety of system-based attacks. Chapter 6 concludes and summarizes.

Chapter 2

Background

This chapter goes over the background necessary to understand this dissertation. A major part of this dissertation is about understanding the different attacks and defenses against memory corruption and Spectre, 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 2.3 explains in detail ARM's Pointer Authentication security mechanism. This is important since it is utilized extensively in the defense mechanisms discussed in this dissertation. Section 2.4 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 [163]. 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.

Executing a control-flow hijacking attack involves abusing memory corruption vulnerabilities. These are prevalent in C and C++ programs, and can lead to critical consequences when exploited by an attacker.

Attackers can abuse many aspects in a program. One popular avenue is maliciously changing

the value of a code pointer, for example, return addresses and function pointers. This is done by corrupting memory that has the pointer and alter the flow of execution of a program [45, 46, 52, 67, 77, 83, 159].

The control-flow hijacking attack in Figure 2.1 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.

2.1.2 Data-Oriented Attack

The main goal of data oriented attacks is to obtain secret information, often in the form of leaking it [57, 93], or arbitrarily execute code.Manipulating data pointers is often the desired attack vector for data-oriented attacks [64]. These types of attacks are more dangerous and powerful than control-flow hijacking attacks. This is mainly due to that they do not manipulate control-data. Numerous secrets have been leaked, such as SSL keys [74, 93]. Defenses are much harder against these attacks, due to the numerous amounts of data pointers that are in a program. In addition to that, defense mechanisms are not able to differentiate between data that is security-sensitive and non-security sensitive ones.

2.2 Spectre attacks

Nowadays, CPUs depend on a variety of optimizations to improve their performance. Some of these optimizations involve executing instructions out-of-order, in order to prevent any

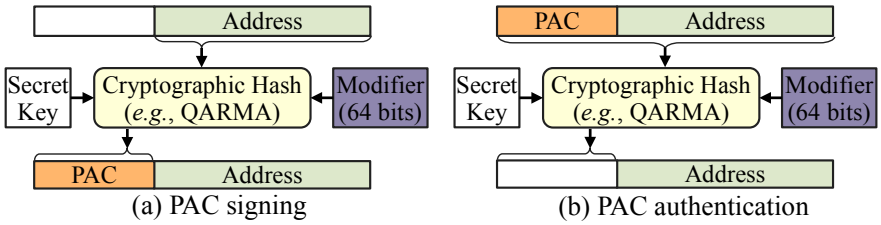


Figure 2.2: PA signs a pointer and generates a pointer authentication code (PAC) based on a address, a secret key, a 64-bit user-provided modifier using PA instructions (*e.g.*, pacia). The signed pointer should be authenticated before the access using the same PAC, address, secret key, and modifier using PA instructions (*e.g.*, autia).

stalling of the pipline when the operands of an instruction are not available. Also, these processors rely on speculation mechanisms to attempt and make predictions of the instruction stream. If a misprediction happens, the resulting execution of instructions is referred to as transient execution. This execution is rolled back architecturally, but its effects are still there in the micro-architectural level, for example in the cache.

A new class of attacks has emerged that abuses speculation mechanisms during transient execution to leak secrets. They are referred to as Spectre attacks, and they abuse predictors that are in the hardware to cause these mispredictions to happen. Instructions are then executed speculatively, in a way that would not have happened if they executed sequentially.

These Spectre attacks abuse the hardware in different ways, depending on which predictor the Spectre attack relies on manipulating. These different types of Spectre attacks are called spectre variants [51, 178]. Several Spectre variants have been discovered. They abuse a variety of speculation mechanisms and hardware predictors [43, 51, 92, 103, 105, 115, 121, 144]. These attacks rely on *accessing* a secret value, and *transmitting* it to a side channel to be leaked. In this dissertation, the focus is on preventing the accessing of the secret.

2.3 ARM Pointer authentication

ARMv8.3-A [95] introduced a new hardware security feature, Pointer Authentication (PA). PA has been implemented in the Apple A12 and M1 chips [173]. Recently, ARM announced ARMv8.6 with Enhanced PAC [161]. 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 [123] 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 [35]. The algorithm takes two 64-bit values (pointer and modifier), as well as a 128-bit key, and gen-

erates 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 4.2(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 2.2(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.

2.4 Current Defenses

2.4.1 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 [95] 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 [30] utilizes a similar scheme for return address protection.

However, because this approach is susceptible to PAC re-use attacks, PARTS return address protection [111] includes the SP with a function ID, generated by PARTS, as a modifier to harden the PAC scheme against re-use attacks. Moreover, PACStack [112] extends the modifier by chaining PACs to bind all previous return addresses in a call stack. On the other hand, PCan [110] 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 [111] 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 [78] 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.

2.4.2 Spectre defenses

Since the leakage of secrets through Spectre attacks is due to changes in the microarchitectural state, a mitigation against Spectre should either prevent the adversary-controlled speculative execution of the accessor from accessing the secret, or prevent the transmitter from speculatively transmitting the secret to the microarchitectural state. Several hardware and software mitigations have been proposed:

Barrier instructions

One way to defend against Spectre is to completely stop speculation whenever a load from memory happens. This is done by placing barrier instructions before the load or after the conditional branch. In the case of x86, this instruction would be Load Fence (lfence) [12]. ARM has Data Memory Barrier (DMB), Data Synchronization Barrier (DSB) [3] and Instruction Synchronization Barrier (ISB) instructions. This method completely prevents Spectre

attacks but incurs significantly high overhead, and completely negates the performance advantages that are offered by speculative execution.

Speculative Load Hardening

Speculative Load Hardening (SLH) [22] prevents the disclosure of data through microarchitectural channels when speculative execution is happening. In order to do that, SLH utilizes a speculation flag and tracks its value. The value depends on the state of speculation. Then, SLH relies on the flag to "harden" the secret and ensure that it does not leak. In the LLVM implementation, the speculation flag is 0 during nominal execution, and is 0xFFF...FFF when misspeculation happens. In order to harden the secret, LLVM-SLH ORs the secret with the speculation flag. This ensures that the secret value is constant during misspeculation and does not leak. In spite of SLH being more efficient than the naive use of barriers, it is still considerably expensive performance-wise for practical use.

2.4.3 Other Defense Approaches

Integrity policies. Control-Flow Integrity (CFI) [24] restricts the valid target sites for indirect control-flow transfers. Static CFI schemes are vulnerable to control-flow bending [53]. Other non-PA dynamic approaches require additional threads to analyze data from Intel Processor Trace [81, 86, 94, 118] limiting scalability.

Code Pointer Integrity (CPI) [106] 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 [50] 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 [73] is to do bounds checking by combining type checking with low fat pointers. TDI [129] relies on grouping types into specific memory arenas by relying on a special allocator and compiler instrumentation. Type after type [168] replaces regular allocations with typed allocations that never reuse memory.

Cryptographic pointer defenses. CCFI [123] 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 [107], DangSan [167], FreeSentry [182], pSweeper [116], and BOGO [187] invalidate all pointers to an object when the object is freed. These schemes typically incur high costs. CRCount [158] implicitly invalidates pointers by using reference counting. This approach comes at memory costs since some objects may never be freed. CETS [132] uses disjoint metadata to check if an object still exists upon pointer dereferences. MarkUs [25] is a memory allocator that protects from use-after-free attacks. It quarantines freed data and prevents reallocation until there are no dangling pointers.

2.5 Past work: PACTight

2.5.1 Motivation

In recent years, the ARM processor architcture started penetrating into the data center [27, 138, 143] and mainstream desktop [28] 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) [24] 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)* [95], 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 [78, 110, 111, 112] and deployed [30, 95], this dissertation identified that they are still exposed to attacks, such as using a signing gadget to forge PACs [36] and reusing PACs [111], allowing arbitrary code execution.

This dissertation proposes PACTight, which is a PA-based defense against control-flow hi-

jacking attacks. In particular, this dissertation 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) *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. This dissertation defines 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.

This dissertation designs 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. This dissertation demonstrates the effectiveness and practicality of PACTIGHT by evaluating with real PA instructions on real hardware.

In summary, this dissertation makes the following contributions:

- This dissertation proposes PACTIGHT, a novel and efficient approach to tightly seal pointers using PAC. By utilizing PACTIGHT's mechanisms, PACTIGHT makes pointers unforgeable, non-copyable, and non-dangling.
- This dissertation implemented four defenses using PACTIGHT: forward-edge protection,

backward-edge protection, C++ VTable pointer protection, and all sensitive pointer protection.

- This dissertation provides a strong security evaluation by demonstrating effectiveness against real-world CVEs and synthesised attacks.
- This dissertation evaluates PACTight implementations on SPEC CPU2006, nbench, CoreMark benchmarks, and NGINX web server with real PAC instructions. 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.

2.5.2 Threat Model and Assumptions

The 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 [98, 128]. Also, the attacker does not control higher privilege levels. PACTight assumes 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 [103], and data only attacks, which modify and leak non-control data, are out of scope. The assumptions are consistent with prior works [68, 110, 111, 112] with the exception of PTAuth [78], which only allows arbitrary write and not arbitrary read.

2.5.3 PACTight Design

In this section, this chapter describes the design of PACTight. This chapter first discusses the design goal (Section 2.5.3), then introduce three pointer integrity properties that PACTight enforces to overcome the limitations of prior PAC approaches (Section 2.5.3),

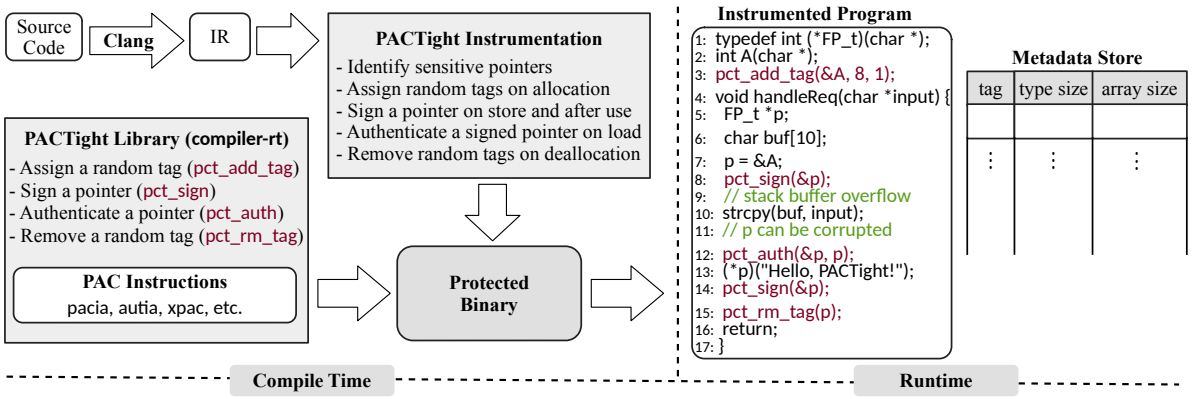


Figure 2.3: PACTIGHT design. At compile time, PACTIGHT instruments the allocation, assignment, use, and deallocation of code pointers and data pointers that are reachable to a code pointer (*i.e.*, sensitive pointers). PACTIGHT guarantees three pointer integrity properties (Section 2.5.3), namely unforgeability, 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 2.5.3). PACTIGHT automates its instrumentation in four different levels: forward edge, backward edge, C++ VTable, and sensitive pointers (§ 2.5.4).

and then compare PACTIGHT to current state-of-the-art defenses (Section 2.5.3). Lastly, present the detailed design of PACTIGHT. As shown in Figure 2.3, PACTIGHT consists of a runtime library and compiler-based instrumentation. This chapter first discusses the runtime (Section 2.5.3) to explain how PACTIGHT enforces the pointer integrity properties and then explain PACTIGHT's automatic instrumentation and defense mechanisms (§ 2.5.4).

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 this goal, it is essential to enforce the complete integrity of pointers, which will be discussed in Section 2.5.3, and prevent any pointer misuse. PACTIGHT protects sensitive pointers [106] – 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

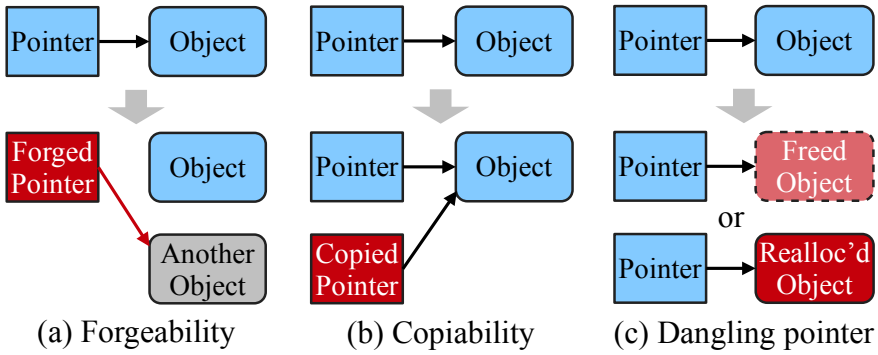


Figure 2.4: Three types of violations of pointer integrity.

summary, the main goals are:

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

PACTight Pointer Integrity Property

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

- **Unforgeability**: As illustrated in Figure 2.4(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 2.4(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 2.4(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 [111] 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 [112] introduces the idea of cryptographically binding a return address to a particular control-flow path by having all previous return addresses in the call stack influence

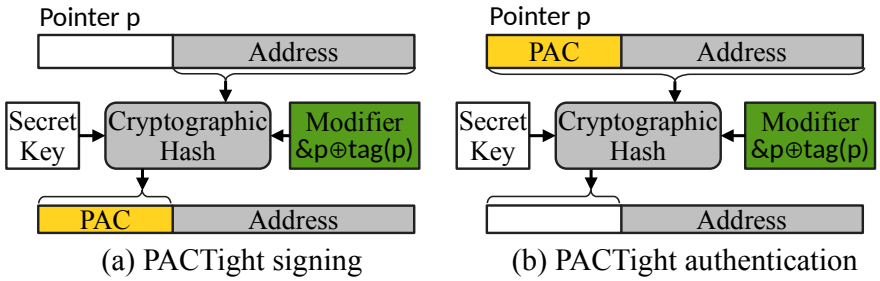


Figure 2.5: Signing and authentication of a pointer variable p in PACTight. In addition to the unforgeability of p provided by PA, PACTight uses the address of a pointer (&p) and a random tag associated with a pointee (tag(p)) to provide the non-copyability and non-dangling properties.

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 [78] 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. This section first describes 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. PACTIGHT proposes 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 2.5.

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

PACTight uses 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 § 2.5.4 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 Figure 2.5.3 (see Figure 2.6). 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, PACTIGHT takes 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

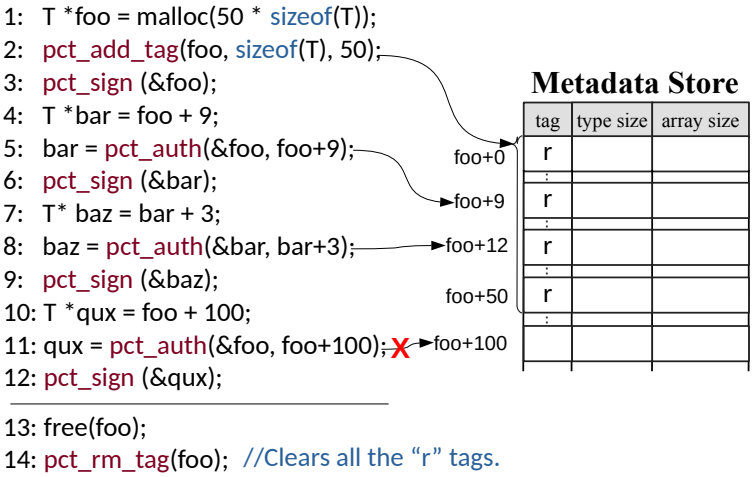


Figure 2.6: Example of PACTight handling array operations.

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 2.6 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 [78] 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. PACTIGHT uses either a 64-bit (default) or a 32-bit tag and the memory overhead is compared between both tag sizes in Section 2.5.6.

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

A Running Example The code snippet in Figure 2.3 (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 it is 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.

2.5.4 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. Four defense mechanisms are implemented: 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. PACTight specially handles 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. PACTIGHT looks 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 [94] 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 [108] 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 [50, 152, 185].

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

Instrumenting PACTight APIs. Upon a C++ type having a virtual function allocated, PACTight-VTable instruments pct_add_tag.It instruments pct_sign immediately after the VTable pointer is assigned by the object's constructor. This adds the PAC to the pointer to seal it. Then, pct_auth is instrumented right before loading the VTable pointer. A failed

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

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 [106]. 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 2.7 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. PACTight over-approximates when detecting security-sensitive pointers. That is, PACTight regards a pointer as security-sensitive if it 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. PACTight aims 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, PACTight uses the *signed* return address of a previous stack frame. Since the stack distance to a return address in a previous

[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.

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 PACTIGHT can achieve the *non-copyability* property for return addresses. In addition, by blending the signed return address of a previous stack frame, PACTIGHT chains all previous return addresses to calculate the PAC of the current return address. This approach is inspired by PACStack [112]. 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. The optimized sign/authentication scheme for return addresses is as follows. The caller's unique function ID is blended with the signed return address from the previous stack frame to generate the modifier. This blending allows us to achieve the *non-copyability* property by chaining all previous return addresses (binding a return address to a control-flow path), alongside the guarantee of the *unforgeability* property achieved by the PAC mechanism. Instrumentation is done in the MachineIR level during frame lowering. Frame lowering emits the function prologues and epilogues. The PAC is added at the function prologue and authenticated at the function epilogue. The LLVM-assigned function ID is unique due to the use of link time optimization (LTO).

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 Figure 2.5.3, 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, PACTight saves 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 the code generation pass prevents the stack spill of the temporary register to avoid the register from being restored.

2.5.5 Implementation

PACTight's 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 § 2.5.4. PACTight uses CSPRNG seeded by hardware RNG (RNDR in ARMv8) [4] for random tag generation. To further harden the prototype, PACTight used different key types for sensitive function pointers (pacia, autia), sensitive data pointers (pacda, autda), and return addresses (pacib, autib).

PACTight has several optimizations to PACTight. First, PACTight uses Link Time Optimization (LTO), which combines all the object files into one file. Then, PACTight inlines all the PACTight runtime library functions. Finally, PACTight implements the additional optimization, discussed in Section 2.5.4, to reduce PAC instructions. The evaluation of the impact of these optimizations is discussed in Figure 2.5.6.

2.5.6 Evaluation

PACTIGHT is evaluated 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? (§ 4.7.1)
- How much performance and memory overhead does PACTIGHT impose? (Section 4.7.2)

Evaluation Methodology

Evaluation environment. All evaluations were ran on Apple's M1 processor [28], which is the only commercially available processor supporting ARMv8.4 architecture with ARM PA instructions. Specifically, an Apple Mac Mini M1 [29] was used equipped with 8GB DRAM, 4 big cores, and 4 small cores. The prototype was ported to Apple's LLVM 10 fork [2]. For all applications, O2 and LTO optimizations were enabled for fair comparison.

Evaluation of C applications. All C applications were ran with real ARM PA instructions. In this case, all Apple LLVM's use of PA [30] are turned off to avoid the conflicting use of PA instructions.

Evaluation of C++ applications. During initial evaluation, PACTIGHT found that the use of PA instructions is built into Apple's standard C++ library. Using Ubuntu Linux [63] on the M1 was investigated 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, two different approaches to validate if PACTIGHT's were used instrumentation is correct and to get an accurate performance estimation. For the correctness testing, all C++ applications were ran on ARM Fixed Virtual Platform (FVP) [32], which

is an ARM hardware platform simulator that supports pointer authentication. The FVP was used only to test correctness, since it is not a cycle-accurate simulator. Linux on FVP was used to run C++ applications, and the Linux kernel and bootloader were modified to activate ARM PA. All the 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, the time to execute a PA instruction was measured 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.* [111] also replaced a PA instruction with four eor instructions to estimate the performance overhead, which is more optimistic than the measurement on hardware.

Security Evaluation

In this section, PACTight's effectiveness in stopping security attacks is evaluated using three real-world exploits (Section 2.5.6) and five synthesized exploits (Section 2.5.6).

Real-World Exploits PACTight was evaluated with three real-world exploits to test its effectiveness against real vulnerabilities.

(1) CVE 2015-8668. This is a heap-based buffer overflow [72] 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 [75] in the libpng [13] 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 identifi-

cation, 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 [154] 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. PACTIGHT was evaluated with five synthesized attacks for C++ to demonstrate how PACTIGHT-VTable can defend against virtual function pointer hijacking attacks, COOP attacks [152] – an attack that crafts fake C++ objects. CFIXX C++ test suite [134] by Burow *et al.* [50] was used. 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, the suite was modified to use a heap-based overflow rather than directly overwriting with memcpy. This modification is similar to a synthesized exploit in OS-CFI [100]. PACTIGHT-VTable detects all the exploits by enforcing pct_auth on the virtual function pointer before the virtual function call. The COOP attack crafts a fake object without calling the constructor and utilizes a virtual function pointer of the fake object. PACTIGHT-VTable detects this due to the fact that it was never initialized and thus pct_auth fails.

Vulnerable code to other PAC defenses. A synthesized exploit is described here that bypasses PARTS [111] and PTAuth [78], 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 2.8. PARTS [111] 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

```
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 2.8: Example of vulnerable code that PACTight defends against but PARTS [111] cannot.

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. The 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 the performance evaluation, three benchmarks are used – namely SPEC CPU2006 [88], nbench [124], and CoreMark [6] – which have been used in prior works, and one real-world application, NGINX web server [23]. In order to run the SPEC CPU2006 benchmark suite, each SPEC benchmark was ported to Apple M1 and built it from scratch. One benchmark, 403.gcc, did not on the Apple M1 even with Apple's vanilla Clang/LLVM compiler. There might be a bug in the MacOS/M1 toolchain. All benchmark applications were ran with real PA instructions except for seven C++ benchmarks in the SPEC benchmark. For the C++ benchmarks, a PA instruction was replaced with seven eor

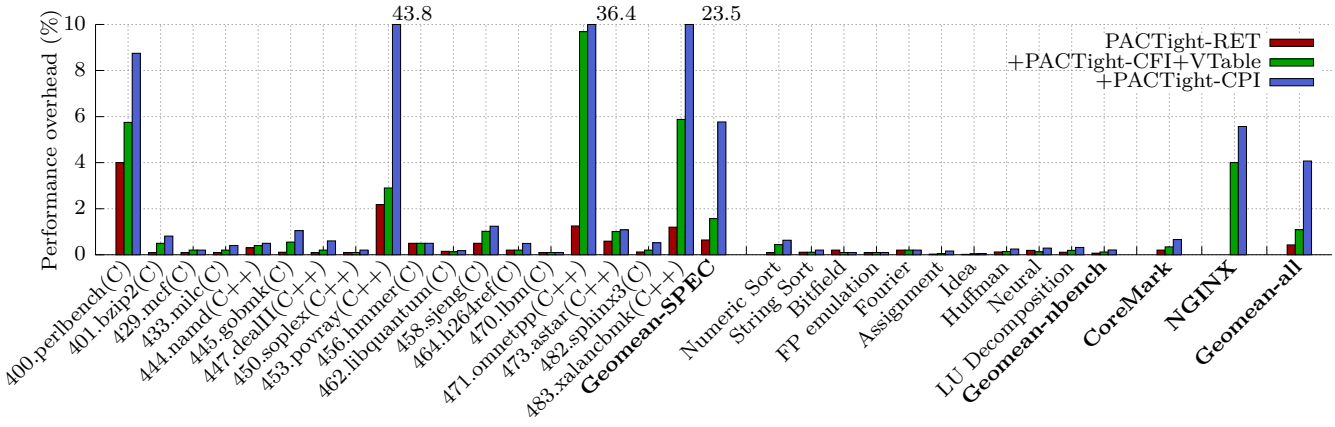


Figure 2.9: The performance overhead of SPEC CPU2006, nbench, and CoreMark relative to an unprotected baseline build. The three PACTIGHT protections are: 1) return addresses (PACTIGHT-RET), 2) CFI, C++ VTable protection and return addresses (+PACTIGHT-CFI+VTable), and 3) CPI offering full protection for all sensitive pointers (+PACTIGHT-CPI).

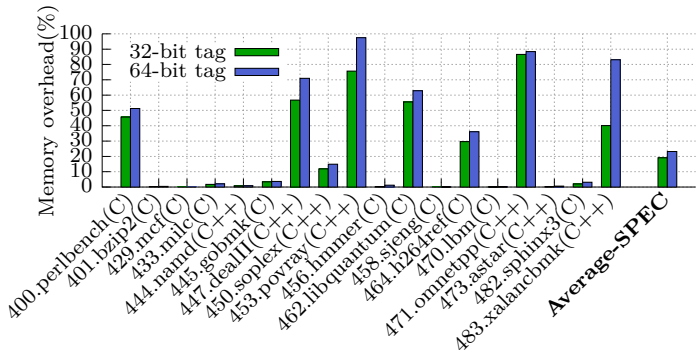


Figure 2.10: The memory overhead of SPEC CPU2006 for PACTIGHT-CPI PACTIGHT's highest protection mechanism. PACTIGHT imposes a low overhead of 23.2% and 19.1% on average for 64-bit and 32-bit tags, respectively.

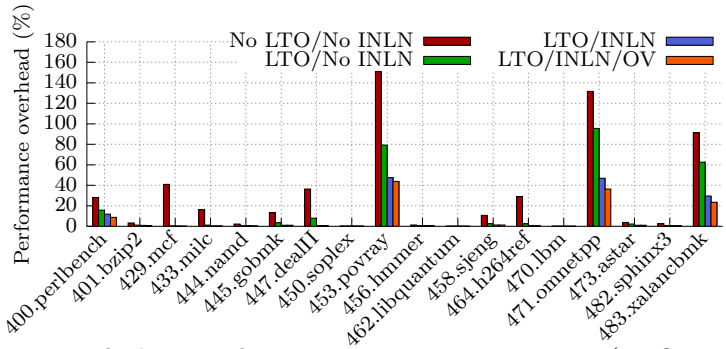


Figure 2.11: Impact of the performance optimizations. (LTO: Link Time Optimization INLN: inlining APIs; OVWRT: overwrite stripped pointer with PACed pointer)

instructions to emulate the overhead of the PA instructions as discussed in Section 3.6.3.

Performance overhead. Figure 4.18 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.

NGINX was evaluated on the Apple M1 using its 4 big cores to stress the machine. The same configuration used to bench NGINX TLS transactions per second [125] was used. The HTTP benchmarking tool wrk [82] was used to generate concurrent HTTP requests. wrk was ran on another machine under the same network. Each wrk spawns three threads where each thread handles 50 connections. A small performance overhead was observed: 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, the maximum resident set size (RSS) was measured during the execution of the SPEC CPU2006 benchmarks. The SPEC benchmarks were ran 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. Both 64-bit and 32-bit tag sizes were used to measure the gain of using 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 2.10 shows the results of the 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 the impact of the optimizations discussed in Section 2.5.4 and Section 4.6 is showcaed. Three optimizations were added 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 2.11 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.

2.5.7 Discussion and Limitations

Information leakage attack on the metadata store. In the 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, 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 [34]. 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 Motivation

Control-flow hijacking and data-oriented attacks have become more sophisticated in recent years. For example, attacks such as DOP [93] and NEWTON [171] 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 [24, 49, 59, 70, 81, 85, 86, 94, 100, 106, 118, 135, 136, 140, 142, 164, 165, 169, 170, 184, 186]. 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 [54, 64, 87, 150]. 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 [151] makes it impractical to protect legacy programs. Moreover, the programmer may not accurately annotate everything, resulting in more possible bugs. Large dynamic metadata [106] 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, the 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, I propose *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.

Runtime Scope-Type Integrity (RSTI) is presented. 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. 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. RSTI-STWC (Scope-Type without Combining) is introduced, which is the 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. Two variants of RSTI-STWC are introduced. 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, the variety of trade-offs between security and performance of RSTI are showcased. To this end, the following contributions are made:

- A new defense policy, Scope-Type Integrity (STI), is introduced 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. STI is the first of its kind to defend against pointer-based attacks by hardening the programmer's intent into machine code.
- Runtime Scope-Type Integrity (RSTI) is proposed, an efficient enforcement of STI using ARM's Pointer Authentication (PA). Three RSTI mechanisms are introduced, RSTI-STWC, RSTI-STC, and RSTI-STL, that instrument all pointers with different restrictions, showing a trade-off between security guarantees and performance overhead.
- All three RSTI mechanisms are prototyped on the LLVM compiler, and demonstrate the feasibility of the proposed schemes on a commercial processor.
- A comprehensive security evaluation of RSTI is provided and its mechanisms on state-of-

```
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 3.1: Data-oriented attack example. The attacker corrupts ptr in Line 4 to bypass the security checks.

the-art control-flow hijacking and data-oriented attacks, as well as real-world CVEs.

- RSTI's mechanisms are evaluated 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

3.2.1 Control-flow Hijacking

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 [45, 46, 52, 67, 77, 83, 159].

3.2.2 Data-Oriented Attack

Data-oriented attacks aim to corrupt non-control data pointers in order to maliciously leak

information [57, 93] 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 [74, 93]. 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 3.1 shows an attack against the GHTTPD web server [58]. 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 [64].

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 3.1, the pointer ptr's scope is the function serveconnection, and should not be used outside that.
- **Permission**: Permissions are 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

The Pointer Authentication feature in ARM is leveraged to enforce RSTI. ARM introduced Pointer Authentication (PA) in ARMv8.3-A [95] 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 4.2(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 4.2(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.

PA is leveraged 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) [15]. 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.

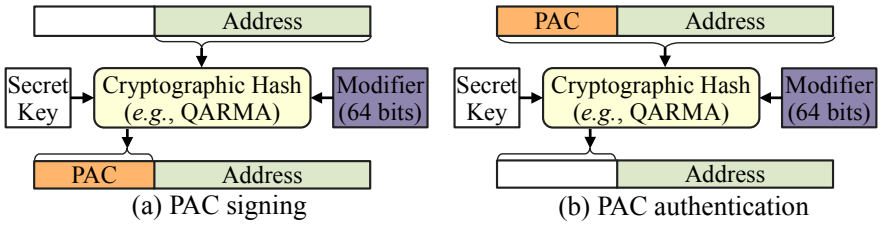


Figure 3.2: PA mechanism signs a pointer and produces a Pointer Authentication Code (PAC) based on the pointer, a user-provided modifier, and a secret key.

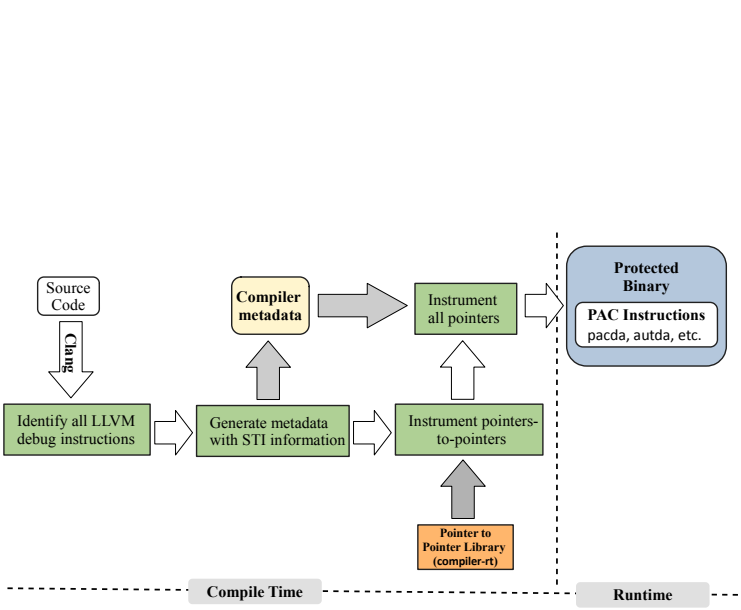
3.3 Threat Model and Assumptions

A threat model of typical memory-corruption attacks [163] is followed. 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. Data Execution Prevention (DEP) is assumed to be in place, and thus the attacker cannot inject their own code. DEP is now enabled by default in most modern operating systems [98, 128]. Also, the hardware and the kernel are trusted; more specifically, that PA keys are securely generated, managed, and stored by the kernel. Attacks that target the kernel and hardware, such as transient execution attacks [103, 147], 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) [55] and other similar techniques [64, 97, 111]. Some techniques that protect data pointers were limited




```
// 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 3.3: 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.

by their reliance on programmer annotation [150]. Other mitigation techniques [97, 106] 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, this dissertation wanted to overcome these challenges. In summary, the 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 is needed, and propagate it to the runtime in some way? If it can be ensured 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. Three main vital pieces of information were identified, 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, this section shows how to defend against a control-flow hijacking attack and data-oriented attack, in Figures 2.1 and 3.1, respectively.

Defending control-flow hijacking. For the control-flow hijacking attack in Figure 2.1, if the enforcement can happen 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 3.1. 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 4.3 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. This dissertation shows that STI is secure enough to ensure that a program executes as the developer intended in § 4.7.1.

- **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, this means 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 3.4: 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 4.3 (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 RSTI checks 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 these types are defined as *compatible types*.

3.4.5 RSTI-types

RSTI-type is defined 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 3.4a 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 3.4a. 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. The RSTI-type is clarified 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.

This dissertation proposes 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 the main RSTI mechanism. RSTI-STWC authenticates and re-signs pointers when casts happen. For example, in Figure 3.4a, 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 3.4a. 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 3.4b, 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 4.7.

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 [97]. 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 3.4a.

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 there is no 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 3.4a.

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 3.5: 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 3.5 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

```
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   ... }
```

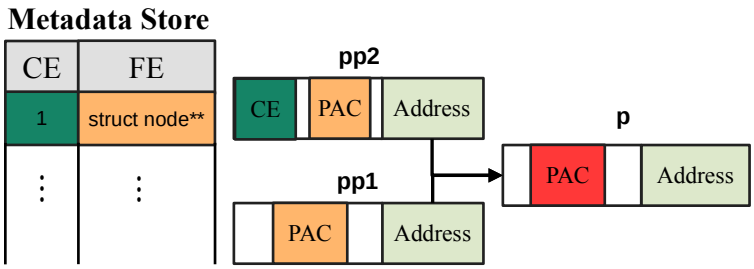


Figure 3.6: Pointer-to-pointer handling to preserve the original type. The Compact Equivalent (CE) and Full Equivalent (FE) refer to a tag and the original type of the pointer-to-pointer, respectively.

from other types that use the !DIDerivedType. Enforcing the scope and type on struct pointer 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, RSTI 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, RSTI needs 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. The 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. RSTI resolves this by adding an extra step of indirection. RSTI defines 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 3.6 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, the 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) [33]. 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, RSTI does 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. The usage of the pointer-to-pointer mechanism is evaluated 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. 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 3.6).
- 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 3.6).
- 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 3.6).
- 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 3.6).

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 3.7. There are two basic

```
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 3.7: RSTI merging of types. The variation allows for different performance and security guarantees.

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 [171]	**Corrupted:** c->send_chain **Target:** malloc	**Type:** ngx_send_chain_pt **Scope:** ngx_http_write_filter	**Type:** void* (size_t size) **Scope:** libc
	AOCR NGINX Attack 1 [149]	**Corrupted:** task->handler **Target:** _IO_new_file_overflow	**Type:** void (*handler) (void *data, ngx_log_t *log) **Scope:** ngx_thread_pool_cycle	**Type:** int *(File *f, int ch) **Scope:** libc
	AOCR NGINX Attack 2 [149]	**Corrupted:** p=log->handler **Target:** ngx_master_process_cycle	**Type:** ngx_log_writer_pt **Scope:** ngx_log_set_levels	**Type:** void *(ngx_cycle_t *cycle) **Scope:** main
	AOCR Apache Attack [149]	**Corrupted:** eval->errfn **Target:** ap_get_exec_line	**Type:** sed_err_fn_t **Scope:** sed_reset_eval, eval_errf	**Type:** char *(apr_pool_t *p, const char *cmd, char * const *argv) **Scope:** set_bind_password
	COOP REC-G [65]	**Corrupted:** objB->unref **Target:** virtual ~Z()	**Type:** class X **Scope:** class Z	**Type:** class Z **Scope:** class Z
	COOP ML-G [152]	**Corrupted:** students[i] ->decCourseCount() **Target:** virtual ~Course()	**Type:** void *() **Scope:** class Student, class Course	**Type:** class Course **Scope:** class Course
	PittyPat COOP Attack [70]	**Corrupted:** member_2->registration **Target:** member_1->registration	**Type:** void*() **Scope:** main, class Student	**Type:** void*() **Scope:** main, class Teacher
	Control Jujutsu NGINX [77]	**Corrupted:** ctx->output_filter **Target:** ngx_execute_proc()	**Type:** ngx_output_chain_filter_pt **Scope:** ngx_output_chain	**Type:** static void *(ngx_cycle_t *cycle, void* data) **Scope:** ngx_execute
	CVE-2014-8668 - libtiff v4.0.6	**Corrupted:** tif->tif_encoderow **Target:** Arbitrary pointer	**Type:** TIFFCodeMethod **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 **Target:** Arbitrary pointer	**Type:** hashfunc **Scope:** inherit_slots, PyObject_Hash	Unknown, since this is a CVE and not an attack
Data oriented attack	DOP ProFTPd Attack [93]	**Corrupted:** &ServerName **Target:** resp_buf, ssl_ctx	**Type:** const char* **Scope:** core_display_file	**Type:** char* **Scope:** pr_response_send_raw
	NEWTON CPI Attack [171]	**Corrupted:** v[index].get_handler **Target:** dlopen	**Type:** ngx_http_get_variable_pt **Scope:** ngx_http_get_indexed_variable	**Type:** void* (const char *filename, int flags) **Scope:** ngx_load_module

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

The prototype is built on top of Apple's LLVM fork [2]. 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.

A few optimizations are applied to RSTI. First, is Link Time Optimization (LTO), which combines all the object files into one file. Also, RSTI inlines all the pointer-to-pointer library functions. In addition to that, the pass is executed 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

RSTI is evaluated by answering the following questions:

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.

- How effective is RSTI in preventing state-of-the-art attacks as well as real-world attacks? (§ 4.7.1)
- 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 programs? (Section 3.6.3)

3.6.1 Security Evaluation

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

Attacks Landscape

In this section, it is shown RSTI can defend against both control-flow hijacking and data-oriented attacks, as well as real-world exploits. Table 4.1 shows a variety of state-of-the-art attacks. RSTI is exercised against powerful synthesized attacks, such as NEWTON [171]

and AOCR [149], as well as C++-specific attacks, such as COOP. Real-world attacks from exploits are also chosen in libtiff and Python. Lastly, RSTI is evaluated against a Data Oriented Programming (DOP) attack [93] that leaks an SSL key.

Attacks in Detail and How RSTI Defends

As Table 4.1 indicates, in all the known attacks, pointers are abused and the change in scope-type information allows the attack to be detected by RSTI. Only two of these attacks are explained 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 it is assumed 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 [93] 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

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

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 prior data pointer integrity works that use PAC, such as PARTS [111], and thus has better coverage against pointer substitution attacks. For example, in the DOP ProFTPD attack in Table 4.1, 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 it is wanted to showcase the usefulness of the RSTI-type information. Let us 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. The number of basic types in each RSTI-type is referred to as *Equivalence Class of Type (EC_T)*. The term *equivalence class* is used 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 it is known, 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. The number of variables in each RSTI-type is referred to 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

It was previously explained the exact use case of pointer-to-pointer handling in Section 3.4.7. The 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 the 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 the intuition that this is a rare case.

3.6.3 Performance Evaluation

Methodology

Evaluation environment. The evaluations were run on the Apple M1 [28], which supports the ARMv8.4 architecture and ARM PA. An Apple Mac Mini M1 [29] was used, that has 4 small cores, 4 big cores, and 8GB DRAM. The prototype was implemented on Apple's LLVM fork [2]. All the applications were compiled with LTO and O2 for fair comparison.

Evaluating C programs. All the C programs were run with real PA instructions. Apple's use of PA [30] was disabled to avoid any conflicts that would happen between RSTI's and Apple's PA instrumentation.

Evaluating C++ programs. Whilst evaluating C++ programs, it was discovered that

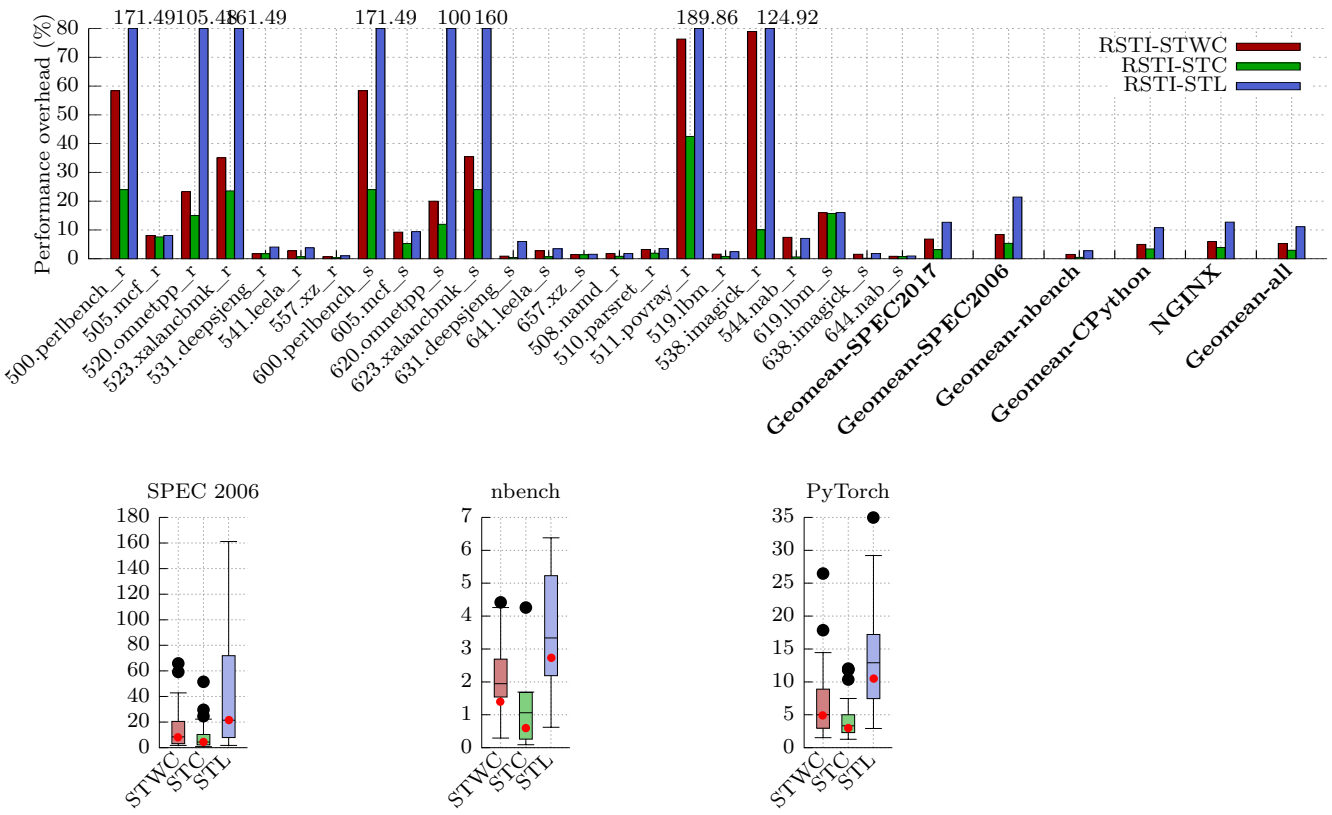


Figure 3.8: The performance overhead of SPEC CPU2017, and the geometric means of SPEC CPU2006, nbench, CPython Pytorch and NGINX for all three RSTI mechanisms. The box plot shows the minimum, median, maximum and quartile values. The black dots represent outlier values. The red dots represent the geometric mean.

PA instructions are built into Apple's standard C++ library. Thus, they could not be turned off. So, for C++ applications, a correctness test was first done on ARM's Fixed Virtual Platform (FVP) [32]. Then, in order to emulate the performance overhead of the PA instructions, seven XOR (eor) instructions were used as an equivalent to one PA instruction on the Mac Mini M1. This has been measured and confirmed in previous works [97, 109, 181].

Benchmarks. For the performance evaluation, a wide variety of benchmarks were used to showcase RSTI's versatility, namely: SPEC CPU 2006 [88], SPEC CPU 2017 [47], and nbench [124]. The SPEC CPU 2006 benchmarks was ported to the Apple M1 and built

them from scratch. 403.gcc and 625.x264 did not run on the M1, in spite of using Apple's own Clang compiler. There is a bug in the macOS toolchain version that was used.

Performance Overhead

Figure 4.18 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, boxplots are used 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 [97].

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. The instrumentation was analyzed 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, PyTorch benchmarks [20] were evaluated 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. NGINX was also evaluated, a real-world application with RSTI on the Apple M1. Its 4 big cores are stressed and use the same configuration used to test NGINX TLS transactions per second [125]. wrk [82] was relied on, 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 [111]. 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, the design choices, such as using LLVM ptrauth intrinsics, running the pass in the backend, using LTO and -O2 optimizations allowed the RSTI compiler to produce much more optimized code. From the correspondence with the PARTS authors, they ran their evaluation with -O0. RSTI's design choices allow us to have lower performance overhead than PARTS whilst

offering higher security guarantees.

3.7 Detailed attacks and benchmarks

In this section, more details are described on the attacks in Table 4.1 and how RSTI defends against them. Also, the showcase the individual performance numbers for SPEC CPU2006, nbench and CPython PyTorch.

3.7.1 Security Evaluation: More details on individual attacks

AOCR NGINX Attack 1 This is an attack on NGINX that performs a malicious write from a buffer to a file in execute-only memory. This necessitates finding functions that open and write files, as well as an indirect call site that has enough arguments to be corrupted. The function chosen is _IO_new_file_overflow which has two arguments. This function is used when a file write is about to overflow its buffer. The attack relies on the indirect call site task->handler in NGINX to call this function, due to both having the same number of arguments. This attack is detected by RSTI due to both pointers being of different types (int *(File *f, int ch) and void (*handler) (void *data, ngx_log_t *log)), as well as being in different scopes. Thus, the scope-type information is different and all three RSTI mechanisms detect this attack.

AOCR NGINX Attack 2 In this attack, the adversary exploits a vulnerability in NGINX to be able to achieve arbitrary code execution and obtain a shell. Similar to AOCR NGINX Attack 1, this attack overwrites an indirect call site (p = log->handler) with a pointer to ngx_master_process_cycle. Based on the analysis, RSTI easily defends this due to the fact that there was no assignment for ngx_master_process_cycle in the source code, and

thus a pointer to it wouldn't have a PAC on it. But even if there is, it would be of type void* (ngx_cycle_t *cycle). This is different from the type of p = log->handler, which is ngx_log_handler_pt. The analysis also shows that they are in different scopes, and thus all three RSTI mechanisms detect this attack.

AOCR Apache Attack For this attack, the adversary reaches the execve system call by corrupting an indirect call site to achieve arbitrary code execution. This is similar to the AOCR NGINX attacks, but this one overwrites the indirect call site with another function that is within Apache. This happens by corrupting the eval->errfn pointer in Apache with a pointer to ap_get_exec_line. Then, when eval->errfn is dereferenced, it jumps to ap_get_exec_line, where the attacker corrupted the arguments and execute a binary of their choice. The analysis found that eval->errfn is of type sed_err_fn_t and ap_get_exec_line is of type char*, and both are in different scopes. Thus, all three RSTI mechanisms detect this attack.

COOP REC-G and ML-G Attacks A counterfeit object-oriented programming (COOP) attack is an attack that crafts fake C++ objects by relying on virtual functions. There are several variants of it. These two specific variants, REC-G and ML-G, rely on recursively calling a destructor function of a different class type to the pointer being corrupted and used to call it. RSTI distinguishes between class types in C++, and generates their scope-type information, similar to types in C. In this case, it is unlikely that a pointer to a destructor function would have a PAC on it at all, and thus authentication would fail. However, even if it did, COOP attacks rely on crafting fake objects of different types, and thus the attack would fail, due to different scope-type information. Thus, all three RSTI mechanisms detect this attack.

COOP PittyPat This attack is also a COOP attack and is similar to REC-G and ML-

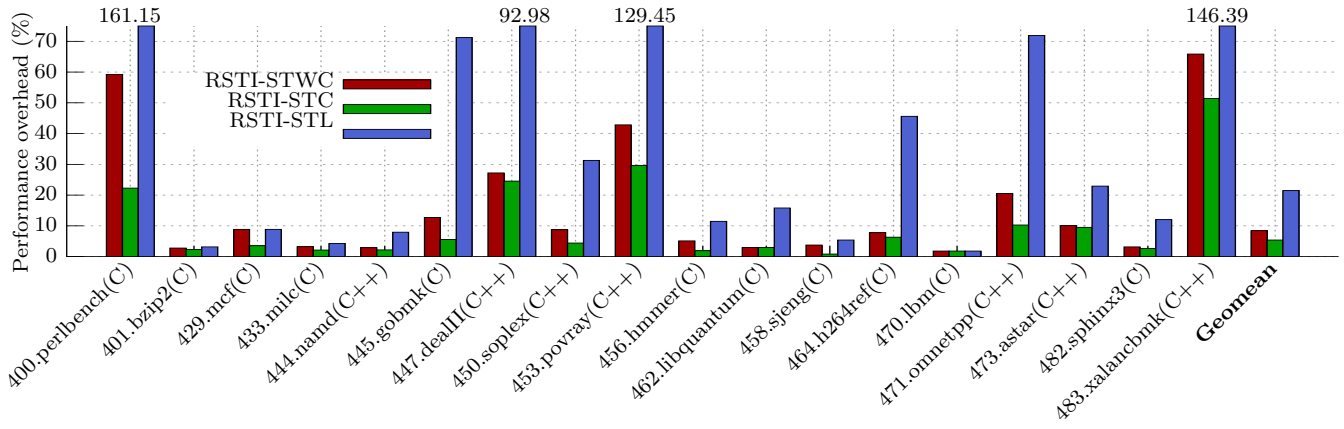


Figure 3.9: The performance overhead of SPEC CPU2006 for all three RSTI mechanisms.

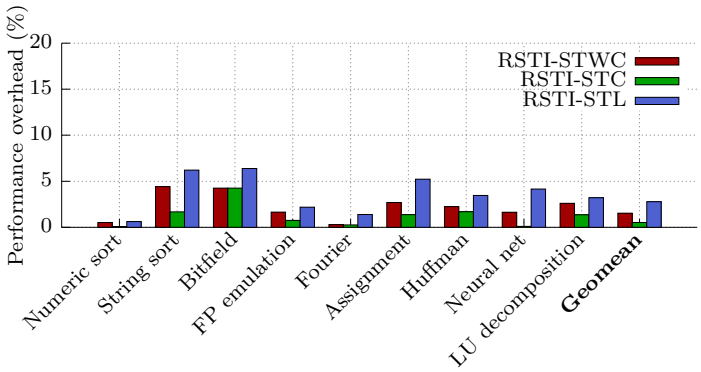


Figure 3.10: The performance overhead of nbench for all three RSTI mechanisms.

G. However, the difference is that this attack does a pointer substitution attack between two class types (Student and Teacher) that both inherit another class (SchoolMember). In case of virtual pointers, the type and scope that RSTI detects is the real type (Student, Teacher) and not the inherited type (SchoolMember). The llvm.dbg information allows for RSTI to detect that.This allows both pointers being substituted to have different scope-type information, and thus RSTI detects this attack.

Control Jujutsu NGINX Attack In this attack, the attacker exploits a vulnerability in NGINX to achieve arbitrary code execution and execute the execve system call. The attacker corrupts the ctx->output_filter pointer to point to ngx_execute_proc, which has the execve

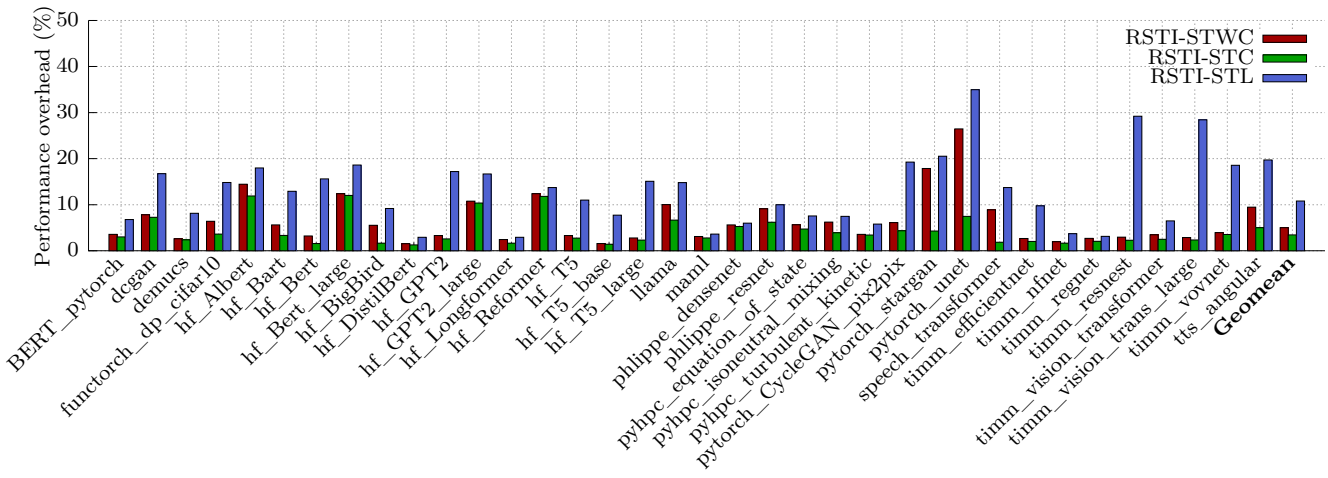


Figure 3.11: The performance overhead of CPython PyTorch for all three RSTI mechanisms.

system call. ctx->output_filter is of type ngx _output_chain_filter_pt and ngx_execute_proc is a void() function and both are in different scopes. In this case, the pointer to ngx_exec ute_proc does not have a PAC, and thus authentication will fail. Even if it did due to a legitimate pointer assignment being present, the scope-type information for both pointers is different and thus all three RSTI mechanisms detect this attack.

CVE-2014-8668 The attacker in this attack overflows the memory due to an unsanitized code following a memory allocation, targeting the TIFF object. The attacker overflows the uncompr_size buffer and targets the pointer tif->tif_encoderow with an arbitrary address. Based on the analysis, there is only one valid assignment for tif->tif_encoderow in the _TIFFSetDefaultCompressionState function, and that is dereferenced in the TIFWriteScanline function. tif->tif_encoderow is of type TIFFCodeMethod. Thus, an attacker would need to conform to this scope-type information if they were to execute an attack. The analysis shows that there aren't any viable pointers for the attack and thus RSTI defends against this attack.

CVE-2014-1912 In this attack, the attacker exploits a missing length check in a buffer to overwrite a pointer in the PyTypeObject. This attack overwrites the tp->tp_hash pointer with

an arbitrary address. The analysis shows that the tp->tp_hash pointer has one valid assignment in the inherit_slots function and is used in the PyObject_hash function. tp->tp_hash is of type hashfunc in Python. In LLVM IR, hashfunc drills down to long*(). Thus, this is the scope-type information captured by RSTI that the attacker would have to conform to in order to carry out a successful attack. This is not achievable based on the analysis and thus RSTI defends against this attack.

NEWTON CPI Attack This is a data oriented attack in which an attacker aims to get to the mprotect system call without modifying any code or data pointers. This attack relies on memory massaging [137] to align the index of a pointer array with a live code pointer to mprotect. This necessitates the usage of a position-independent executable (PIE) being randomly placed by ASLR and repeated attempts. This attack corrupts op->data in NGINX, which reaches the index in the function pointer array v[index].get_handler. op->data is corrupted beyond the size of v[index] in which the pointer called out of bounds reaches mprotect. NEWTON uses an available dlopen, from ngx_dlopen, to call dlopen on "/bin/ed" or other shared objects that use the system library call and force the linker to bind the system code pointer in memory with the global offset table (GOT). Then, NEWTON corrupts v[index].get_handler again to call the new live system pointer.

For this attack, there are two avenues of defense for RSTI. First, a pointer to ngx_dlopen is available in NGINX in the handle pointer in the ngx_load_module function. This pointer is of type void* (const char *filename, int flags).Thus, this scope-type information allows RSTI to restrict the abuse of the pointer pointing to ngx_dlopen. Since v[index].get_handler is used to maliciously call ngx_dlopen, their scope-type information must match. v[index].get_handler is of type ngx_http_get_ variable_pt which boils down to int*() in the LLVM IR, and it is used in the ngx_http_get_indexed_variab le. Thus, their scope-type information is different and RSTI would detect the attack. However, for RSTI-STC, the int*() pointer casts to a void*

somewhere in NGINX and so those types are compatible. RSTI-STWC and RSTI-STL defend this attack at this stage.

Second, for the second stage of the attack, which consists of calling v[index].get_h andler beyond its intended scope to the live system code pointer, that pointer must have the same scope-type information as v[index].get_handler. Since this pointer is from libc, the scope-type information will be different, or that pointer may not even have a PAC on it. In both cases, RSTI defends against the attack this stage. This allows RSTI to enforce deterministic spatial and temporal memory safety, due to the compartmentalization of scope-types between pointers. This is discussed more in Section 4.8.

3.7.2 Performance Evaluation: Detailed numbers

In this section, the detailed performance numbers are shown for SPEC CPU2006, nbench and CPython PyTorch. SPEC 2006 benchmarks have a geometric mean of 8.42%, 5.36%, and 21.47% for RSTI-STWC, RSTI-STC, and RSTI-STL respectively, with some benchmarks like perlbench, povray, and xalancbmk being exceptionally high, due to heavy usage of pointers. The nbench benchmarks have a geometric mean of 1.54%, 0.52%, and 2.78% for RSTI-STWC, RSTI-STC, and RSTI-STL respectively. 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.

3.8 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 this is relegated 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 the 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 § 4.7.1.

Is the PAC length enough? In macOS, and in the 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 [109] 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 1.52×10^{-5}. This is considered to be negligible, and does not give an attacker practicality to commit an attack [91]. As for the 8 bits in the case of pointer-to-pointer, this allows for 256 possible PACs, and it is 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 the evaluations, the default Apple return address protection scheme is turned on.

3.9 Related Work

Type-based defenses. The goal of EffectiveSan [73] is to do bounds checking by combining type checking with low fat pointers. TDI [129] relies on grouping types into specific memory arenas by relying on a special allocator and compiler instrumentation. Type after type [168] replaces regular allocations with typed allocations that never reuse memory.

ARM PAC defenses. PARTS [111] protects pointers with pointer authentication by relying on the LLVM ElementType as the modifier. AOS [101] utilizes the PAC as an index to access a bounds table and do bounds checking.

Data oriented defenses. Data Flow Integrity (DFI) [55] makes sure that a variable can only be written by its legitimate write instruction. Hardware assisted Data Flow Isolation (HDFI) [160] provides instruction-level isolation by relying on tags. YARRA [151] 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) [24] constricts the number of valid target sites for an indirect control-flow transfer. Code Pointer Integrity (CPI) [106] protects a subset of pointers, referred to as sensitive pointers.

3.10 Summary

This dissertation 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. RSTI was presented, 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. Three RSTI defense mechanisms were implemented with varying levels of security granularity, enforcing control-flow and data-flow integrity. The security of RSTI was showcased 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.

Chapter 4

Mitigating Spectre with sPACtre

4.1 Motivation

Speculative execution is a micro-architectural optimization technique in which a processor executes certain instructions in advance by anticipating that their results may be needed in the near future. In other words, the processor ”speculates” that these instructions will be executed soon. If the speculation is correct, the instructions have already been executed and the processor can use the results sooner. If the speculation is incorrect, these results are squashed architecturally. Thus, speculative execution brings significant performance benefits.

However, speculative execution introduces security issues, as the attacker can trigger the processor to execute arbitrary instruction sequences (speculatively) in the program, deviating from the control flow of the designed software code, as shown in Figure 4.1. Even though the results of the speculative execution are no longer available architecturally, they are still available in the micro-architecture states, such as in the cache. This phenomenon has led

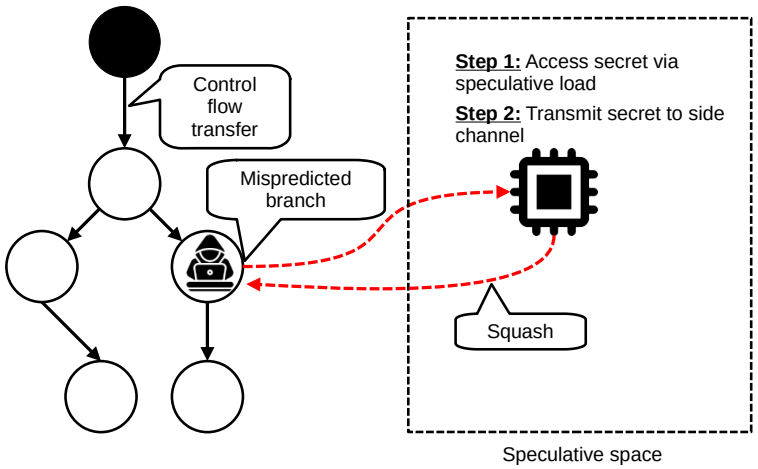


Figure 4.1: Spectre control-flow attack. Misprediction of a conditional branch, target branch or return address happens by poisoning the Pattern History Table (PHT), Branch Target Buffer (BTB) or the Return Stack Buffer (RSB)

to to Spectre attacks [103, 127], a class of attacks that abuse speculative execution to leak possible secrets through micro-architectural side channels. To leak secrets, Spectre attacks rely on abusing a *Spectre gadget*, a series of instructions that give the attacker the ability to obtain and leak information via speculative execution. Because there are many possible ways to trigger and manipulate speculative execution and many possible microarchitecture side channels, there have been many Spectre attack variants [43, 51, 90, 92, 102, 103, 105, 114, 121, 144, 147, 176, 178].

In response, many defenses have been announced that aim to address either one or several Spectre variants [3, 10, 11, 12, 21, 22, 51, 60, 62, 66, 80, 102, 131, 133, 153, 157, 172, 174, 178, 183, 188]. These defenses aim to prevent the secret from being *accessed* speculatively during a misspeculation, or prevent the Spectre gadget from *transmitting* the sensitive information to the side channel. A technique that many software defense mechanisms rely on is the use of speculation barriers [3, 12]. Speculation barriers are instructions that prevent speculation until all instructions before them finish execution. Barriers are effective against Spectre, since they effectively stop speculation, but using them naively can be extremely costly, incurring performance penalties of up to 25x in some benchmarks [133]. Many defense mechanisms

aim to minimize the use of them as much as possible. Intel-lfence [12] adds barriers after each conditional branch, however, it does not protect spectre attacks that are not due to the conditional branches. Blade [172] further reduces the barrier instructions with dataflow analysis using a static type system, but it only covers one of the Spectre variants.

In addition to that, the majority of Spectre state-of-the-art software defenses are implemented in the x86 architecture, with some of them relying heavily on Intel-specific hardware security features such as Intel-CET [96] and Intel-MPK [126]. Swivel [133] focuses on hardening WebAssembly against Spectre attacks, and relies on the use of Intel-MPK, as well as the shadow stack and forward-edge control flow protection offered by Intel-CET to build its defense mechanism. Serberus [131] also relies on Intel-CET to restrict transient control-flow to an extent that it can statically identify specific Spectre taint primitives that leak secrets and add lfences to them to prevent them from leaking secrets. Serberus and Swivel are both efficient due to their reliance on Intel-CET to eliminate the gadgets that can be used by the attacker. This presents a challenge when wanting to adopt these mechanisms for other architectures such as ARM. ARM does not have a full Intel-CET equivalent.

Some other Spectre defense mechanisms need hardware modification and are not deployable on existing hardware. ProSpeCT [66] defends against several Spectre variants, but is a RISC-V implementation and thus is not deployable on existing hardware. ARM has been rising in popularity outside of the embedded market, and has fully infiltrated the desktop [28] and data center market [27, 138, 143]. Spectre attacks have been shown to be effective on ARM as well [51]. Thus, a mitigation mechanism that can be deployed on ARM today is needed.

To this end, SPACTRE is proposed, a novel mitigation mechanism that aims to prevent the speculative *access* of secrets in a Spectre gadget. Since SPACTRE focuses on the Spectre gadget, SPACTRE covers Spectre attacks regardless of what type of speculative execution and regardless of the side channel. SPACTRE achieves this effectively and efficiently by

relying on a combination of software defense mechanisms and existing hardware security features (e.g., PAC) on ARM.

Spectre gadgets are usually comprised of two components:

- **Accessor:** An instruction that accesses a secret value or the value to be leaked, and reads it into the architectural state due to speculative execution controlled by the adversary.
- **Transmitter:** An instruction that transmits the secret value or the value to be leaked over a convert channel

sPACTre aims to prevent the malicious speculative access, or load, of a secret in the *accessor*. sPACTre utilizes ARM's Pointer Authentication mechanism [95]. Pointer Authentication is a mechanism that aims to enforce pointer integrity by relying on a cryptographically computed Pointer Authentication Code (PAC) that gets placed on the top bits of the pointer. sPACTre relies on PACs to ensure that pointers cannot be tampered with by an attacker and speculatively load values in a load that is not supposed to load a secret. To prevent secret loads from being abused, sPACTre relies on annotated secrets in the program to add access restrictions on all the loads. This is composed of using a different modifier for secrets to generate the PAC, and adding a barrier instruction at a legitimate secret load. This guarantees that an attacker cannot abuse other non-secret loads to access the secret, and cannot speculatively execute the legitimate secret load. To prevent attackers from abusing out-of-bounds array accesses to leak secrets at arbitrary addresses, sPACTre has its own dynamic tagging mechanism for arrays that makes sure arrays cannot be accessed outside its bounds to defend against out-of-bounds accesses. The tagging mechanism relies on tagging every array element with a tag, and checking if the tag exists on every array access at the index. If the array element does not have a tag, the top two bits of the address are flipped, thus the contents of it are inaccessible. This happens speculatively and thus allows for a Spectre attack to be defended without relying on barrier instructions for the leaking array.

A security analysis on SPACTRE is performed, as well as a performance evaluation. This dissertation shows that SPACTRE is effective in defending against Spectre attacks that leverage speculative execution of instruction sequences, including, Spectre-PHT [103] that targets the branch prediction mechanism to cause speculative execution at branches, Spectre-BTB[103] that poisons the Branch Target Buffer (BTB) and abuse the branch target predictor speculatively, Spectre-RSB [105] that manipulates the Return Stack Buffer (RSB) and relies on misspeculation based on return addresses, and Spectre1.1 [102] that uses a speculative store to manipulate control flow.

The SPACTRE compiler pass is implemented and evaluated SPACTRE on a variety of Constant Time (CT) [26] cryptographic programs, such as libhydrogen [9], libsodium [14], and OpenSSL [17]. Several algorithms from these programs are evaluated, such as AES, ChaCha20 [41], and Curve25519 [42]. SPACTRE is also evaluated with a synthetic benchmark from a prior Spectre defense mechanism [80]. The results show that SPACTRE is effective and efficient in preventing Spectre attacks. Thus, this dissertation makes the following contributions:

- SPACTRE is proposed, a new defense mechanism to prevent the malicious speculative load (access) of a Spectre gadget on ARM by utilizing ARM Pointer Authentication, annotation of secrets and a dynamic tagging mechanism for checking array bounds.
- The SPACTRE prototype is implemented on the LLVM compiler architecture and showcase its feasibility on a commercial processor. Thus, SPACTRE is deployable on current hardware.
- SPACTRE is showcased on how it can defend against Spectre attacks, such as Spectre-PHT, Spectre-BTB and Spectre-RSB attacks.
- A full performance evaluation of SPACTRE is provided on a variety of cryptographic algorithms in several cryptographic libraries, as well as a synthetic benchmark, and show the

efficiency of sPACtre in these scenarios.

4.2 Background

4.2.1 Spectre attacks

Modern processors rely on optimizations to improve performance. To improve the execution time, *speculation mechanisms* predict execution flow and start execution speculatively even when it is unclear whether the execution would happen. When the prediction is mostly correct, the speculation mechanisms allow the execution to start early without stalling the pipeline. *Transient execution* is the execution of instructions that results from a misprediction. Transient execution is reverted architecturally, however its effects on the micro-architectural level, such as the cache, are still there.

Spectre attacks [103] take advantage of the speculation mechanisms to leak secrets when transient execution happens. Speculation can be at the execution unit, where some values are used speculatively in the circuits [48, 92, 115, 166]. Speculation can also be at the instruction level, such that certain instruction sequences are executed. sPACtre focuses on the latter. The speculation in the execution unit heavily depends on the processor design and many of them only affect certain processor designs, e.g., cyclic return stack buffers is only in Intel processors not in ARM. Speculating execution instructions enables the attacker to weaponize existing instruction sequences in the software code to access secrets that it otherwise is not able to access. Such instruction sequences used in speculative attacks are known as Spectre gadgets. Spectre gadgets are mainly composed of two components:

- **Accessor:** The accessor's main goal is to perform a speculative load on a secret value, or the value that the attacker aims to leak. This is done to write the secret into a micro-

architectural component, such as the cache, so that it can be leaked. The use of the accessor is powerful to an attacker, due to the fact that no actual change happens in the program and naively limiting the accessor hinders speculative execution, which severly impacts performance.

- **Transmitter:** After the accessor loads the secret value speculatively, this secret value needs to be leaked to the attacker. This is done by the transmitter instruction. The transmitter relies on a covert channel that can read the secret value from the microarchitectural component, such as the cache.

Different spectre attacks exploit different hardware predictors. These are referred to as *spectre variants* [51, 178]. There have been many spectre variants discovered, and they exploit a variety of hardware predictors and speculation mechanisms [43, 51, 92, 103, 105, 115, 121, 144]. In this work, the following spectre variants are considered, which cause speculative execution of instructions:

Spectre-PHT. Spectre-PHT [103] exploits the Pattern History Table (PHT) which is used to predict the result of conditional branches. This attack poisons the PHT to mispredict the direction of conditional branches (taken or not taken). Since conditional branches are used by programmers or compilers to maintain certain memory safety invariants, the Spectre-PHT attack is used by attackers to bypass control-flow integrity or memory safety isolation guards by abusing the wrong-path execution. Spectre V1 [103] is an attack in which the attacker poisons the PHT in order to leak data out-of-bounds speculatively. Spectre1.1 [102] also poisons the PHT but relies on overwriting the return address to steer control-flow.

Spectre-BTB. In Spectre-BTB [103], the attacker poisons the Branch Target Buffer (BTB). The BTB is used to predict the target of an indirect jump. This causes transient execution to happen on a mispredicted branch target. Spectre-BTB is similar in concept to Return Oriented Programming (ROP), since it can redirect the speculative control flow to an arbi-

trary location. It relies on polluting the BTB and can be used to execute a ROP-style attack by speculatively executing gadgets that are not in the normal control-flow path [44].

Spectre-RSB. Spectre-RSB [105, 121] abuses the Return Stack Buffer (RSB). The RSB is a microarchitectural buffer, for each core, that stores the virtual addresses after most recent call instructions. When a ret instruction is executed, the CPU pops the top element in the RSB in order to predict the return flow. The attacker can manipulate the stack so that the actual return address stored on the stack does not match the one on the RSB. By controlling the call address, the attacker has control of the misspeculation address.

4.2.2 Covert channel

A covert channel is a means to exfiltrate the secret from the microarchitectural state. For example, cache-based covert channels are commonly used, in which an attacker speculatively leaks the secret to the cache[117, 139, 141, 179] and identifies the leaked information by timing cache accesses. The Translation Lookaside Buffer (TLB) [84] can also be used as a covert channel. Prime+Probe [117], Evict+Reload [113], and Flush-Reload [179] are attack strategies that are commonly used.

4.2.3 ARM Pointer Authentication

ARM introduced Pointer Authentication in ARMv8.3-A [95] and is available in commercial processors [28]. Pointer Authentication is a hardware security feature that can be utilized to protect the integrity of pointers. This is done by generating a Pointer Authentication Code (PAC) with a cryptographic hash algorithm. The cryptographic algorithm takes as input the pointer, a secret key, and a 64-bit modifier. The modifier is specified by the developer.

The default cryptographic hash algorithm used is the QARMA [35] algorithm, but this can be modified to use another custom algorithm. Apple does that in its implementation of Pointer Authentication [30]. In order to sign the pointers, the algorithm takes as input the pointer, modifier, and key. It then generates a PAC that is placed on the top unused bits of the pointer, as shown in Figure 4.2(a). In order to authenticate the pointers, the algorithm takes as input the pointer with the PAC on it, as well as the modifier and key that were used to sign the pointer. The PAC is then recomputed and verified with the one on the pointer. If they match, the PAC is taken off the upper bits, as shown in Figure 4.2(b). If they do not match, the top two bits of the pointer are flipped, rendering the pointer to be unusable. Pointer Authentication uses pac instructions for signing pointers and aut instructions for authenticating, with distinct instructions depending on the type of pointer and the secret key.

The LLVM compiler infrastructure has pointer authentication intrinsics, referred to by llvm.ptrauth [15]. These are intrinsics internal to LLVM that allow Pointer Authentication instructions to be utilized directly from the compiler, without relying on an external library. llvm.ptrauth.auth and llvm.ptrauth.sign intrinsics take three arguments:

- pointer address: This is the raw address that is to be signed. This must be passed as a 64-bit integer. SPACTRE uses LLVM's PtrToInt instruction to convert the pointer type to an integer. Then, after the signing or authenticating process is done, SPACTRE uses the IntToPtr instruction to convert the integer value back to a pointer.
- key: This is an integer value that tells the intrinsic which key will be used. There are 5 possible key values, representing 5 different instructions for authenticating and signing. The last two characters in the assembly instruction name show which key is used and what type of pointer, *e.g.*, autdb means an authentication instruction for data pointers using the B-key.

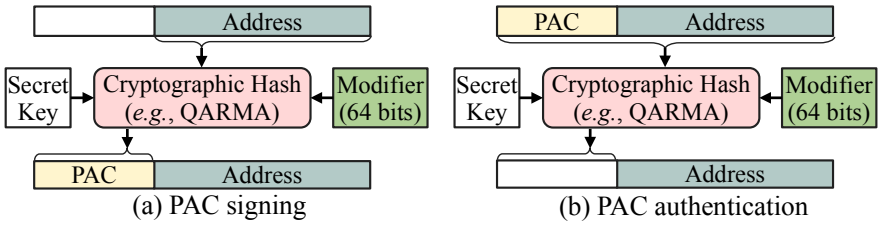


Figure 4.2: The Pointer Authentication mechanism signs a pointer and generates a Pointer Authentication Code (PAC) based on the pointer, a secret key and a modifier provided by the user. The pointer must be authenticated with the same parameters (pointer, secret key, and modifier) that were used when signing.

- modifier: The modifier is a 64-bit integer that adds diversity to the PAC generated. sPACtre utilizes this to generate different PACs based on whether or not the variable is marked as secret.

4.2.4 Spectre mitigations

Since the leakage of secrets through Spectre attacks is due to changes in the microarchitectural state, a mitigation against Spectre should either prevent the adversary-controlled speculative execution of the accessor from accessing the secret, or prevent the transmitter from speculatively transmitting the secret to the microarchitectural state. Several hardware and software mitigations have been proposed:

Barrier instructions. One way to defend against Spectre is to completely stop speculation whenever a load from memory happens. This is done by placing barrier instructions before the load or after the conditional branch. In the case of x86, this instruction would be Load Fence (lfence) [12]. ARM has Data Memory Barrier (DMB), Data Synchronization Barrier (DSB) [3] and Instruction Synchronization Barrier (ISB) instructions. DMB is a memory barrier that guarantees that all memory accesses that are in the program order before the DMB instruction are architecturally executed before any memory accesses that appear in the program order after the DMB instruction. ISB flushes the pipeline in the processor. This

means that all instructions following the ISB instruction are fetched from memory or cache, after the instruction has been completed. ARM recommends using both DMB and ISB, and that using DMB alone does not completely prevent speculative execution [90]. SPACTRE did not use ARM's Consumption of Speculative Data Barrier (CSDB) instruction due to the fact that it was shown to be vulnerable [90]. This method completely prevents Spectre attacks but incurs significantly high overhead, and completely negates the performance advantages that are offered by speculative execution.

Speculative Load Hardening. Speculative Load Hardening (SLH) [22] prevents the disclosure of data through microarchitectural channels when speculative execution is happening. In order to do that, SLH utilizes a speculation flag and tracks its value. The value depends on the state of speculation. Then, SLH relies on the flag to "harden" the secret and ensure that it does not leak. In the LLVM implementation, the speculation flag is 0 during nominal execution, and is 0xFFF...FFF when misspeculation happens. In order to harden the secret, LLVM-SLH ORs the secret with the speculation flag. This ensures that the secret value is constant during misspeculation and does not leak. In spite of SLH being more efficient than the naive use of barriers, it is still considerably expensive performance-wise for practical use.

Retpoline. Retpoline [21] replaces all the indirect branches with return trampolines that utilize an infinite loop which never gets executed to prevent the CPU from speculating on the target. Retpoline only defends against Spectre-BTB and has been shown to be prone to attacks that target return instructions [176].

Intel eIBRS. Intel has proposed some mitigations to Spectre including enhanced Indirect Branch Restricted Speculation (eIBRS) [10]. Enhanced Indirect Branch Restricted Speculation (eIBRS) is a branch control defense mechanism that adds speculation restrictions on indirect branches by guaranteeing that indirect branch prediction cannot be controlled by a

software program that gets executed in a lower-privilege predictor mode. eIBRS also targets only defending against Spectre-BTB and doesn't prevent the other Spectre variants.

Intel CET - Indirect Branch Tracking (IBT). Intel CET [96] is an extension to the instruction set architecture that helps protect against Jump Oriented Programming (JOP) [46] and Return Oriented Programming [155]. One of the mechanisms Intel CET offers is Indirect Branch Tracking (IBT) which allows for the enforcement of coarse-grained Control Flow Integrity (CFI) [24] to protect against JOP. IBT relies on using a branch termination instruction, endbranch. Programs that want to use IBT's protection place endbranch at all valid indirect call targets. If the indirect call instruction jumps to another instruction, the processor triggers a control-flow violation. This mechanism is also effective during speculation, in which the speculation is halted.

Intel CET - Shadow Stack. Intel CET also offers a hardware-based shadow stack to protect against ROP attacks. This shadow stack is used to check and maintain the integrity of return addresses on the running program's stack. The shadow stack is not accessible via store and load instructions, and thus cannot be tampered with. Similar to IBT, Intel CET's shadow stack halts speculation when it detects any violations. One thing to note is that ARM does not have a hardware-based shadow stack implementation.

ARM BTI. ARM introduced Branch Target Identification (BTI) in ARMv8.5-A [5] to help defend against Jump Oriented Programming (JOP) attacks [46]. BTI instructions are inserted at the start of a function and are designated as landing pads. The processor is configured so that indirect branch instructions (BR and BLR) can only be allowed if the landing pad is present. If the target of the indirect branch is not a landing pad, the processor would throw an exception. The use of BTI reduces the number of gadgets an attacker may jump to, limiting a JOP attack from being executed. This is similar to Intel CET's Indirect

Branch Tracking (IBT) [96].

4.2.5 Comparison with other defenses

Table 4.1 shows the comparison between SPACTRE and other state-of-the-art Spectre mechanisms. The table shows the category of the defense, what architecture it is implemented on, whether or not it is deployable, and what Spectre attacks it defends against. As can be seen, the majority of the defenses are on the Intel architecture, with little that are implemented for Arm. Also, the majority do not cover all the Spectre attacks shown. v1.1 is listed separately from PHT in the table. This is because v1.1 can modify the return address speculatively, and thus steering the control flow to a Spectre gadget, like in BTB/RSB. Recently, there have been annotation-based Spectre defense mechanisms such as ConTExT [153] and ProSpeCT [66]. However, these both rely on custom hardware, and thus aren't deployable. SPACTRE adds a defense mechanism for the ARM architecture, is deployable on existing hardware and defends against all the shown Spectre attacks.

Table 4.1: Different Spectre defense mechanisms and how they compare to SPACTRE. Arch. stands for Architecture and D. stands for Deployable.

Category	Mitigation	Arch.	D.	PHT	BTB	RSB	V1.1
Software Mitigations	Intel-lfence	x86	✓	✓	×	×	×
	LoadBarrier	x86/ARM	✓	✓	✓	✓	×
	Retpoline	x86	✓	×	✓	×	×
	oo7	x86	✓	✓	×	×	✓
	Blade	x86	✓	✓	×	×	✓
	LLVM-SLH	x86/ARM	✓	✓	×	×	×
	UltimateSLH	x86	✓	✓	×	×	✓
	Swivel	x86	✓	✓	✓	✓	✓
	Serberus	x86	✓	✓	✓	✓	✓
Hardware Mitigations	Intel eIBRS	x86	✓	×	✓	×	×
	CSV2	ARM	✓	×	✓	×	×
Annotation Mitigations	ConTExT	Simulation	×	✓	✓	✓	✓
	ProSpeCT	RISC-V	×	✓	✓	✓	✓
	SPACTRE	ARM	✓	✓	✓	✓	✓

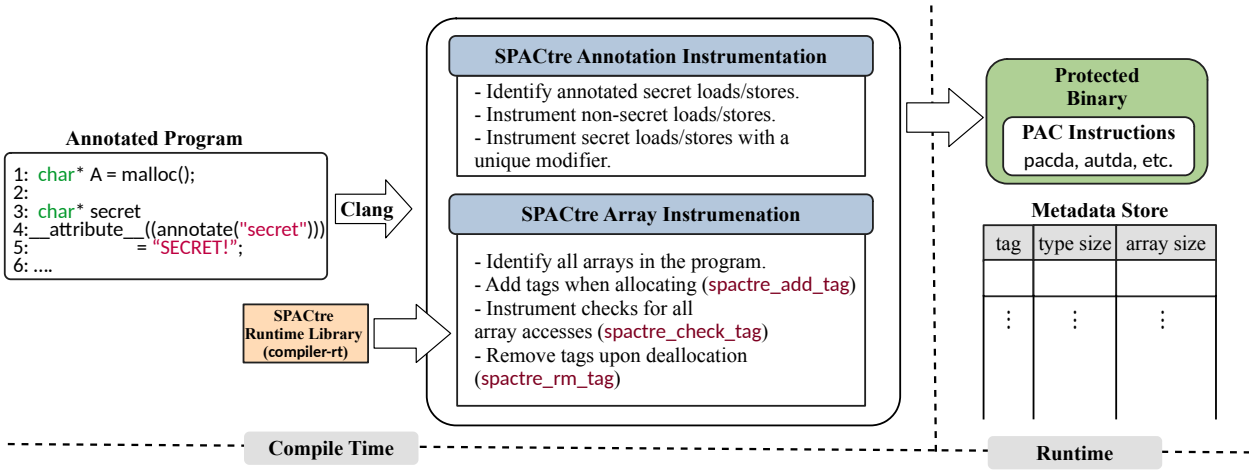


Figure 4.3: Overview of sPACtre.

4.3 Threat Model and Assumptions

The OS and the victim's program are assumed to both be logically correct, and that the attacker and the victim program are compiled with the sPACtre compiler. Thus, the code used by the attacker is instrumented with sPACtre's protections. In addition to that, Data Execution Prevention (DEP) is enabled, and thus the attacker is unable to inject their own code. DEP is enabled by default now in the majority of modern operating systems [98, 128]. In addition to that, the Pointer Authentication secret keys are generated, stored, and managed by the kernel securely. Memory corruption attacks are considered to be orthogonal to sPACtre. Other protections against memory corruption attacks can be deployed together with sPACtre.

The goal of the attacker is to learn the secret data contents by means of speculative execution attacks. Also, the attacker is assumed to be able to leak data via a microarchitectural covert channel on the machine that is running the victim program, and that the attacker has access to a timer that they can use to probe and measure latencies of microarchitectural events.

4.4 sPACtre's Design

The main goal of SPACTRE is to defend against Spectre attacks efficiently and effectively by preventing the speculative *access* to secrets in a program. In other words, SPACTRE aims to prevent the accessor in the Spectre gadget from maliciously loading the secret speculatively. SPACTRE aims to do this by reducing the number of barrier instructions used, to achieve good performance, while completely preventing the attacks.

Also, current state-of-the-art Spectre defenses are either unimplemented hardware-oriented (custom hardware or hardware modifications) [60, 183], or are a combination of hardware modifications and software [66, 153], or are software only [131, 133, 157, 172, 174, 188]. There are also implemented hardware mitigations by hardware vendors [3, 10, 12, 21]. The majority of these are in the Intel architecture, with little solutions for ARM. SPACTRE aims to have a robust and effective software solution for ARM that can run on existing hardware. In summary, the main goals are:

- **Prevent the speculative access of a secret:** Effectively defend against the malicious speculative load of a secret.
- **Minimal usage of barriers:** Reduce the number of barrier instructions and only use them where necessary.
- **Compatibility for ARM:** No need for hardware modifications and can run on existing ARM hardware.
- **High performance:** Low runtime overhead.

4.4.1 Design Philosophy

Spectre attacks can be defended against by either preventing the secret value from being accessed speculatively by means of a malicious misspeculation, *i.e.*protecting the accessor, or by preventing the transmission of the secret value over the covert channel when malicious misspeculation happens, *i.e.*protecting the transmitter.

Observation 1: Load barriers are effective protection but come with high performance overhead. Hardware vendor white papers have stated that the use of barriers is one of the most effective ways to prevent Spectre attacks [11]. However, naive use of barriers incurs a significant performance penalty. Thus, many state-of-the-art solutions [133, 172, 174] have made design decisions to try and minimize the use of barriers while still defending against Spectre. Other mechanisms [10, 21, 22] have tried to eliminate barriers completely and use an alternative method. However, these mechanisms have been shown to be vulnerable [176] or still incur a high performance penalty [22]. In addition to that, barriers by themselves are not definitively secure. Spectre1.1 [102] overwrites the return address and steers control-flow to bypass the barrier instrumentation. Thus, combining barriers with other defense techniques is needed.

Observation 2: Without correct authentication of the PAC on a pointer, the load of that pointer in both regular and transient execution is prevented. Since the existence of a PAC on the pointer changes the pointer semantics, the PAC must be authenticated properly before the pointer can be used. A dereference of the pointer with the PAC on it will not be successful. This happens in both regular and transient execution. Thus, in order for an attacker to successfully execute a Spectre attack, they need to be able to obtain a valid, authenticated pointer.

Observation 3: PAC encodes protection metadata in the pointers effectively and

efficiently. Pointer Authentication relies on the unused bits to store the metadata, and these get propagated with the pointer. This allows the protection of pointers to be enforced effectively and efficiently. It is believed that the use of ARM's Pointer Authentication, combined with other techniques, to minimize the use of barrier instructions to only where necessary is an effective approach. This is what is aimed to be achieved with SPACTRE. SPACTRE aims to efficiently and effectively prevent the malicious speculative access that is done by a Spectre gadget by relying on ARM's Pointer Authentication feature, programmer annotation of secret variables, barrier instructions, and a tagging mechanism for arrays. The use of ARM's Pointer Authentication to instrument all loads and the tagging mechanism for arrays to check array bounds reduce the need to insert barrier instructions to only loads where the secret variable is annotated.

Observation 4: Programmer annotations help reduce protection overhead. Since the main goal of Spectre attacks is to leak a secret, other state-of-the-art solutions have used programmer annotations to annotate the secret data [66, 153]. This allows for a more focused defense and would achieve the same overarching goal as a more automated solution. The trade-off here is that this needs more effort from the programmer, but the advantage would be a more efficient defense. One thing to note is that the two state-of-the-art defenses that use annotations, ProSpeCT [66] and ConTExT [153], both rely on hardware modifications for their main implementations and thus are not readily deployable.

4.4.2 Design Overview

SPACTRE makes sure that the accessor is not able to speculatively leak secrets to the micro-architectural level. This is done by protecting all the loads in a program with a PAC, relying on programmer annotations to protect secret loads with a PAC that has a modifier unique

to secrets and a barrier instruction. The barrier instruction prevents the secret from being leaked, and the PAC with a different modifier prevents the attacker from abusing other loads to leak the secret. sPACTre also defends against Spectre-PHT attacks by enforcing array bounds at runtime. The bounds checking mechanism stores a tag for every array element in a dynamic metadata in memory. This allows Spectre-PHT attacks to be prevented without relying on a barrier instruction at the conditional branch.

Figure 4.3 shows the overall design of sPACTre. The input to sPACTre is a program with the secrets annotated by the developer. At compile time, all stores are instrumented with PAC signing instructions and all loads are instrumented with PAC authenticating instructions. The secret loads use a modifier unique to them and different from all other loads. The secret loads also have a barrier instruction inserted after the authentication instruction. In addition to that, arrays are instrumented with functions from the sPACTre runtime library to set the tags for array elements, check the tags whenever an array element is accessed, and remove tags when the lifetime of the array is done.

4.4.3 Annotation of secrets

sPACTre aims to prevent speculative loads of secrets. In order to prevent secrets from being leaked, sPACTre needs to know which variables hold these secret values. Thus, sPACTre relies on the programmer to annotate the program's secret variables. sPACTre only requires the programmer to add ___attribute___((annotate("secret"))) to the variable's declaration. Figure 4.3 shows an example of how this annotation would look like. sPACTre is then able to track all the memory accesses associated with it. All loads/stores that are secret use a different modifier when generating the PAC. This means that the attacker cannot reuse valid PACs to leak the secret.

Annotating Constant Time (CT) programs.

Constant Time programming is an approach to defend against hardware side-channel leaking in software [41, 42, 79, 130, 156, 175]. CT programs are written in way so that secrets are not supplied to transmitters in all sequential executions. The CT model has been widely utilized in cryptographic code [14, 17, 189]. Thus, several techniques and tools support CT code verification [26, 40] and generation [56, 69, 175]. Previous work also surveys these approaches [39].

Even though annotation may be cumbersome for some programs, it was observed that certain programs are easier to annotate. SPACTRE focuses particularly on Constant Time (CT) cryptographic programs. It was observed that annotating the secrets of a specific cryptographic algorithm in one of these programs is not difficult or time consuming. This is due to the fact that secrets in CT programs are well defined and identifiable. In addition to that, the constrictions enforced by the CT programming model prevents secrets from being used in many locations, and thus it is able to identify and properly annotate secrets without a significant effort. This is discussed more and evaluate SPACTRE on CT programs in Section 4.7.2.

4.4.4 Preventing speculative PAC signing

SPACTRE instruments all pointer assignments in a program with PAC instructions. This means that at all times the pointers have PACs on them and are protected. SPACTRE utilizes LLVM's pointer authentication intrinsics for instrumentation [15]. The intrinsics allow authentication to be done without spilling to memory, due to compiler optimizations. Thus, there is no need to sign pointers again after authentication, since the memory accesses are optimized by the compiler and the authenticated address of the pointer is always in a

register.

Some speculative execution attacks abuse stores speculatively [43, 44]. Since sPACTre uses a different modifier between secrets and non-secrets, a possible avenue to abuse a non-secret pointer assignment is to speculatively sign a secret pointer address with a non-secret modifier and then load it. This would succeed, since the non-secret load does not have a barrier. sPACTre prevents this by adding a barrier instruction after all PAC signing instructions. This means that the attacker cannot speculatively abuse PAC signing.

4.4.5 Protecting the speculative accessor

A Spectre attack can jump anywhere speculatively, depending on which Spectre variant is used. This means that all the loads in a program are potentially abused by Spectre attacks as the accessor. sPACTre instruments all loads in a program with PAC instructions, and thus sPACTre protects the speculative accessor. There are several ways that sPACTre protects the accessor, depending on how it is accessed.

Direct pointer access

An attacker may try to leak data by an accessor that is the load of an address (without pointer arithmetic) speculatively.

If the load in the accessor is originally accessing a non-secret, then the PAC authentication will pass only if the pointer has the correct PAC with the non-secret modifier. sPACTre has a different modifier for secret pointers, and thus, the attacker cannot use non-secret pointers to leak secrets, because an attacker cannot reuse valid PACs between pointers [97] when the modifier is different.

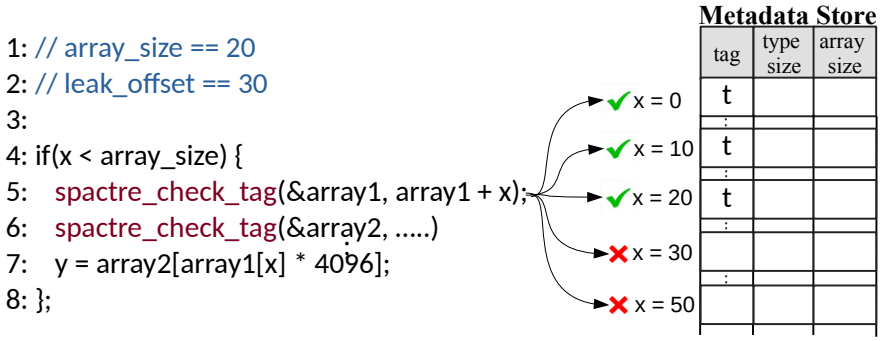


Figure 4.4: SPACTRE tag checking mechanism for arrays.

If the load in the accessor is originally intended to access a secret, SPACTRE prevents the malicious access by inserting a Data Memory Barrier (DMB) and Instruction Synchronization Barrier (ISB) instructions after the authentication. These barriers prevent the secret load from being executed speculatively.

Arrays and the tag checking mechanism

An attacker may try to abuse arrays in a Spectre attack. Spectre-PHT is an example of that. Spectre-PHT leaks the secret by abusing a load that is not the secret load. Spectre-PHT relies on accessing a secret memory buffer from another memory buffer that is not secret. The two buffers can be next to each other or at an offset known to the attacker. The attacker speculatively overflows the non-secret buffer to access the secret buffer by abusing the conditional bounds check in the code.

A possible solution would be to add a barrier at the conditional bounds check to prevent the overflow from happening speculatively. However, adding a barrier at every conditional would be costly, as shown by prior work [131, 178]. Thus, a different approach for SPACTRE is adopted.

One of the goals is to reduce the number of barriers added as much as possible. SPACTRE adopts a **tag checking mechanism** for arrays. A metadata in memory is initialized that

stores tags for every array element in the program. These tags are stored at an offset from the original array element address. Whenever an array is initialized, sPACTre initializes its corresponding metadata. Whenever an array is accessed, sPACTre checks to see if a tag exists in the metadata for that specific array element location. If there isn't, sPACTre flips the top two bits of the address, rendering it to be unusable. This happens only speculatively in the case of a Spectre-PHT attack, and thus the address is unusable at the microarchitectural level when the overflow happens. This prevents the secret from being leaked.

Figure 4.4 showcases how the mechanism works. The metadata store is implemented as a linear open addressing hash table (base + offset) using the address as the key. The base address is kept in the X18 register, which is reserved to avoid a stack spill from leaking the metadata location. The metadata is maintained by sPACTre 's runtime library.

4.4.6 Enforcing the protection mechanism

It is needed to make sure that the protection mechanism executes before all the loads, and that the attacker cannot skip the protection mechanism. sPACTre prevents an attacker from bypassing any of its instrumentation. This is ensured by limiting the start point of the speculative execution. For forward-edge control-flow speculation, sPACTre limits the attacker from jumping past the security checks by enabling ARM BTI [5]. BTI instructions (bti) are added to the start of each indirect call target. If the attacker tries to jump to the middle of a function in an attempt to bypass the security checks, execution will be halted. For backward-edge control-flow speculation, sPACTre adds tag checking (spactre_check_tag) whenever an array store is done, which prevents the attacker from abusing return addresses and array stores out-of-bounds to speculatively sign and store secrets before the loads get executed, like in Spectre1.1 [102]. In the Apple M1 ARM chip, the RSB gets cleared on

overflow [16], and thus will return to a function return, which will not be in the middle of an PAC authentication and load/store.

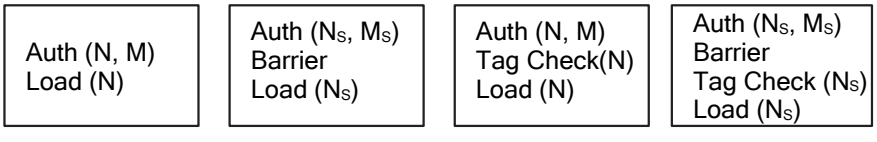
4.4.7 Security analysis

SPACTRE instruments loads in four ways:

- **Non-secret and non-array loads:** These only have the PAC authentication instruction instrumented before them.
- **Secret and non-array loads:** These have the PAC authentication instruction followed by a barrier instrumented before them.
- **Non-secret and array loads:** These have the PAC authentication instruction followed by a metadata tag check (spactre_check_tag) instrumented before them.
- **Secret and non-array loads:** These have the PAC authentication instruction followed by a barrier followed by a metadata tag check (spactre_check_tag) instrumented before them.

Figure 4.5 shows how all four look like. The SPACTRE compiler is able to detect these and add instrumentation accordingly. The main advantage of having these is that this allows us to minimize the need of using barrier instructions. Figure 4.11 in § 4.5.3 shows the LLVM IR for all of them.

AS discussed in Section 4.4.6, trying to bypass the authentication, barrier or tag check



(a) Non-Secret Non-Array (b) Secret Non-Array (c) Non-Secret Array (d) Secret Array

Figure 4.5: SPACTRE instrumentation of loads.

and execute the load directly is not possible. This is due to the existence of the PAC. The PAC needs to be authenticated and the attacker cannot bypass the authentication instruction. Also, the attacker cannot execute their attack by executing the barrier or tag check. Executing at the barrier means that the speculation will not happen and thus the attack is halted. Trying to bypass the barrier and executing at the tag check also stops the attack, the Spectre-PHT attack specifically, due to the fact that the tag check prevents out-of-bounds-access.

In the following sections, it is analyzed conceptually how sPACTre defends against the Spectre control-flow attacks, but a more detailed evaluation is in § 4.7.1.

Defending against abusing a Spectre gadget with secret access

Spectre-BTB and Spectre-RSB are Spectre attacks that rely on the Spectre gadget to leak secrets without relying on pointer arithmetic. This means that these attacks do not abuse bounds and attempt to leak the secret directly from the secret load. sPACTre is able to defend against these attacks.

Defending against Spectre-BTB. sPACTre defends against the Spectre-BTB attack by relying on the barrier instruction to prevent the secret from being leaked. Spectre-BTB relies on redirecting the control-flow speculatively to a function that leaks a secret by poisoning the branch predictor. Since the secret used in that secret function is annotated, a barrier instruction will be inserted whenever a memory access to it happens. The existence of the barrier instruction stops the attack. In addition to that, the attacker will not be able to leak the secret via a different memory load due to the existence of the PAC that has a unique modifier for secret variables.

Defending against Spectre-RSB. sPACTre is able to stop Spectre-RSB similarly to

Spectre-BTB. Spectre-RSB also relies on a secret load happening speculatively due to a malicious modification of the software stack. This tricks the RSB to leak the secret to the cache. Due to the fact that this secret will be annotated by the programmer, a barrier instruction will be inserted and the attack will be stopped. Also, similar to Spectre-BTB, the attacker will not be able to use another memory load to leak the secret due to the unique PAC.

Defending against abusing a Spectre gadget with pointer arithmetic

Spectre-PHT is a Spectre attack that leaks the secret through a Spectre gadget, but not directly through the secret load. Instead, it relies on abusing another buffer and overflowing it by relying on pointer arithmetic to leak the secret. SPACTRE is able to defend against this attack.

Defending against Spectre-PHT. SPACTRE is able to defend against Spectre-PHT by relying on the tag checking mechanism. The mechanism checks tags when arrays are accessed. The violation is detected when the array goes out of bounds speculatively, since the tag check is also executed speculatively. When the check fails, the top two bits of the address are flipped and so the pointer is rendered unusable and will not result in a speculative leakage. Thus, SPACTRE is able to defend against Spectre-PHT without requiring the insertion of a barrier at the conditional branch.

Summary

SPACTRE is able to prevent the malicious speculative load of the accessor in the Spectre gadget. SPACTRE relies on PAC to harden the code and limit the loads that an attacker can abuse, due to the different modifier for secret values. SPACTRE relies on the tag checking

mechanism for arrays to eliminate the need for adding barrier instructions to conditional branches. Thus, sPACtre only limits the insertion of barrier instructions to the secret loads. The attacker cannot execute the secret load speculatively due to the barrier. The combination of all these techniques restricts the attacker and allows the defense to be effective and efficient. sPACtre defends against the Spectre-PHT, Spectre-BTB and Spectre-RSB attacks.

4.5 Extended and instrumented code snippets

4.5.1 Extended code snippets for the attacks

This section showcases more extended code snippets for the PoCs of Spectre V1, Spectre V2 and Spectre-RSB attacks that were used to evaluate the security of sPACtre. These code snippets show the annotation of secret, but the assembly shown is without the instrumentation. The annotation of secret is added here to showcase how the annotation would like when applied to code. The code in Figure 4.6 shows the extended C code and ARM assembly for the Spectre V1 PoC. The code in Figure 4.7 shows the extended C code and ARM assembly for the Spectre V2 PoC. The code in Figure 4.8 shows the extended C code and ARM for the Spectre-RSB PoC. Due to the nature of the Spectre-RSB attack that was implemented, there is have a mix of inline assembly code and C code. This is due to the fact that inline assembly was needed to be able to manipulate the stack.

4.5.2 sPACtre instrumented assembly code for the attacks

This section shows the instrumented assembly for the instrumented LLVM IR in Figure 4.13 and Figure 4.16 for Spectre V1 and Spectre V2, respectively.

```
1  // C-code
2
3  // Pointer to public data.
4  char* array1;
5
6  // Secret data
7  // leak_offset = secret - array1
8  char* secret __attribute__((annotate("secret")));
9
10 // Array accessed by victim.
11 char* array2;
12
13 void victim(){
14 ...
15   if(x < array1_size) {
16     y = array1[x]; //accessor
17     z = array2[y]; //transmitter
18   }
19 ...
20 }
21
22 // Assembly
23 victim:
24         sub     sp, sp, #32
25         ldr     w8, [sp, #28]
26         ldr     w9, [sp, #24]
27         subs    w8, w8, w9
28         b.ge    .LBB0_2
29         b       .LBB0_1
30 .LBB0_1:
31         ldr     x8, [sp, #8]
32         ldrsw   x9, [sp, #28]
33         ldr     w8, [x8, x9, lsl #2] // accessor load
34         str     w8, [sp, #20]
35         ldr     x8, [sp]
36         ldrsw   x9, [sp, #20]
37         ldr     w8, [x8, x9, lsl #2] // transmitter load
38         str     w8, [sp, #16]
39         b       .LBB0_2
40 .LBB0_2:
41         add     sp, sp, #32
42         ret
```

Figure 4.6: Spectre V1 code with annotation of secret. The assembly is not instrumented.

The instrumented assembly of the victim function is shown in Figure 4.9 from the Spectre-PHT PoC (Figure 4.6). The authentication instruction is showcased (Line 16 and Line 25) as well as the instrumentation for spactre_check_tag for bounds checking (Line 20 and Line

```
// C code
// Indirect branch that will be used to poison
// the branch target predictor.
void (**attack)(int);
// Array accessed by the victim.
char* array2;
// Secret data accessed by the secret function.
char* secret __attribute__((annotate("secret")));
// Dummy data accessed by the dummy function.
char* dummy;
// On training, attack will call this function.
void secret_fun(int x){
        y = secret[x];
        z = array2[y];}
// On misprediction, attack will call this function.
void dummy_fun(int x){
        y = dummy[x];
        z = array2[y];}
// Called during training and misprediction.
void victim(void (**func)(int), size_t offset) {
    ((void (*)(int))func[0])(offset);}
// Assembly
secret_fun:
        ldr     x8, [x8, :lo12:secret]
        ldrb    w8, [x8]
        adrp    x9, y
        strb    w8, [x9, :lo12:y]
        ldr     x8, [x8, :lo12:array2]
        ldrb    w9, [x9, :lo12:y]
        add     x8, x8, x9
        ldrb    w8, [x8]
        adrp    x9, z
        strb    w8, [x9, :lo12:z]
        add     sp, sp, #16
        ret
victim:
        ldr     x8, [sp, #8]
        ldr     x8, [x8]
        ldr     x9, [sp] // func
        mov     w0, w9   // offset
        blr     x8       // mistrained branch
        ldp     x29, x30, [sp, #16]
        ret
```

Figure 4.7: Spectre V2 code with annotation of secret. The assembly is not instrumented.

31).

For the Spectre V2 PoC (Figure 4.7), the instrumented assembly of the secret_fun function

```
// Secret attacker is trying to leak
char* spectre_secret __attribute__((annotate("secret")));
// Used as spectre gadget
static uint8_t spectre_buffer[256 * 256];
// Use a whole page for return address
// so there is nothing closeby
void* return_addr_arr[2048];
// Function that returns to address pointed
// to by argument instead of return address
// void called_function() {
//     x1 = &spectre_buffer[256]
//     x30 = return_addr_arr[1024];
//     return /* to x30 */;}
asm(
".global called_function\n"
"called_function:\n"
"    adrp x1, _spectre_buffer@PAGE\n" // prepare gadget
"    add x1, x1, _spectre_buffer@PAGEOFF\n"
"    add x1, x1, #0\n"
"    adrp x0, _return_addr_arr@PAGE\n"
"    add x0, x0, _return_addr_arr@PAGEOFF\n"
"    add x0, x0, #8192\n"
"    ldr x30, [x0]\n" // x30 = return_addr_arr[1024].
"    ret\n"
);
// void gadget() {
//    return_addr_arr[1024] = x30;
//    thrash_cache(); // Randomly access a large array
//    called_function();}
void gadget(uint64_t idx){
  asm(
  ".global gadget\n"
  "gadget:\n"
  "    adrp x0, _return_addr_arr@PAGE\n"
  "    add x0, x0, _return_addr_arr@PAGEOFF\n"
  "    add x0, x0, #8192\n"
  "    mov x1, #0\n"
  "    str x30, [x0]\n" // return_addr_arr[1024] = x30
  "    bl _thrash_cache\n"
  "    bl called_function\n"
  );
  // called_function does not return here.
  while(1) {
    // Executed speculatively.
    // Encode secret into cache.
    // Only executed during misprediction
    *(spectre_buffer + spectre_secret[idx] * 256);}}
```

Figure 4.8: Spectre-RSB code with annotation of secret. The assembly is not instrumented.

```
1  victim:
2          sub     sp, sp, #64
3          stp     x29, x30, [sp, #48]
4          add     x29, sp, #48
5          mov     w8, #24
6          str     w8, [sp, #4]
7          ldursw  x8, [x29, #-4]
8          ldr     x9, [sp, #8]
9          ldr     x9, [x9]
10         subs    x8, x8, x9
11         b.hs    .LBB1_2
12         b       .LBB1_1
13 .LBB1_1:
14         ldr     x8, [sp, #24]
15         ldr     w9, [sp, #4]
16         autda   x8, x9
17         str     x8, [sp, #24]
18         ldr     x0, [sp, #24]
19         ldur    w1, [x29, #-4]
20         bl      spactre_check_tag
21         ldr     x8, [sp, #24]
22         ldursw  x9, [x29, #-4]
23         ldr     w8, [x8, x9, lsl #2]
24         stur    w8, [x29, #-12]
25         ldr     x8, [sp, #16]
26         ldr     w9, [sp, #4]
27         autda   x8, x9
28         str     x8, [sp, #16]
29         ldr     x0, [sp, #16]
30         ldur    w1, [x29, #-4]
31         bl      spactre_check_tag
32         ldr     x8, [sp, #16]
33         ldursw  x9, [x29, #-12]
34         ldr     w8, [x8, x9, lsl #2]
35         stur    w8, [x29, #-16]
36         b       .LBB1_2
37 .LBB1_2:
38         ldp     x29, x30, [sp, #48]
39         add     sp, sp, #64
40         ret
```

Figure 4.9: Spectre V1 assembly code for victim instrumented with sPACTre.

is shown in Figure 4.10. Similar to the instrumented assembly of the victim function in the Spectre V1 PoC, shows the authentication instruction (Line 11) and spactre_check_tag (Line 17). However, the instrumented barrier instructions (Lines 13 and 14) are shown.

```
1  secret_fun:
2          sub     sp, sp, #32
3          stp     x29, x30, [sp, #16]
4          add     x29, sp, #16
5          stur    w0, [x29, #-4]
6          adrp    x8, secret
7          str     x8, [sp]
8          ldr     x9, [x8, :lo12:secret]
9          adrp    x10, mod
10         ldr     w10, [x10, :lo12:mod]
11         autda   x9, x10
12         str     x9, [x8, :lo12:secret]
13         dmb     sy
14         isb
15         ldr     x0, [x8, :lo12:secret]
16         ldur    w1, [x29, #-4]
17         bl      spactre_check_tag
18         ldr     x8, [sp]
19         ldr     x8, [x8, :lo12:secret]
20         ldursw  x9, [x29, #-4]
21         add     x8, x8, x9
22         ldrb    w8, [x8]
23         adrp    x9, y
24         strb    w8, [x9, :lo12:y]
25         adrp    x8, array2
26         ldr     x8, [x8, :lo12:array2]
27         ldrb    w9, [x9, :lo12:y]
28         add     x8, x8, x9
29         ldrb    w8, [x8]
30         adrp    x9, z
31         strb    w8, [x9, :lo12:z]
32         ldp     x29, x30, [sp, #16]
33         add     sp, sp, #32
34         ret
```

Figure 4.10: Spectre V2 assembly code for secret_fun instrumented with SPACTRE.

4.5.3 LLVM IR of sPACtre's instrumentation of loads

Figure 4.11 shows a snippet of the LLVM IR for all four ways that SPACTRE instruments loads, discussed in § 4.4.7. The variable secret is annotated early in the program when it gets declared and SPACTRE is able to add the barrier instrumentation to its loads. In addition to that, SPACTRE relies on the GEP instruction (arrayidx) to identify arrays, as well as their allocation.

```
// Non-Secret and Non-Array
%1 = load i8*, i8** non-secret, align 8
%auth.pi = ptrtoint i8* %1 to i64
%2 = call i64 llvm.ptrauth.auth(i64 %auth.pi, i32 2, ...
%3 = inttoptr i64 %2 to i8*
%4 = load i8, i8* %3, align 1

// Secret and Non-Array
%1 = load i8*, i8** secret, align 8
%auth.pi = ptrtoint i8* %1 to i64
%2 = call i64 llvm.ptrauth.auth(i64 %auth.pi, i32 2, ...
%3 = inttoptr i64 %2 to i8*
call void asm "DMB SY\0AISB SY", ""()
%4 = load i8, i8* %3, align 1

// Non-secret and Array
%1 = load i8*, i8** non-secret, align 8
%auth.pi = ptrtoint i8* %1 to i64
%2 = call i64 llvm.ptrauth.auth(i64 %auth.pi1, i32 2, ...
%3 = inttoptr i64 %2 to i8*
%4 = load i64, i64* %index, align 8
%5 = bitcast i8* %3 to i8**
call void @spactre_check_tag(i8** %5, i64 %4)
%arrayidx = getelementptr inbounds i8, i8* %3, i64 %4
%6 = load i8, i8* %arrayidx, align 1

// Secret and Array
%1 = load i8*, i8** secret, align 8
%auth.pi = ptrtoint i8* %1 to i64
%2 = call i64 llvm.ptrauth.auth(i64 %auth.pi1, i32 2, ...
%3 = inttoptr i64 %2 to i8*
call void asm "DMB SY\0AISB SY", ""()
%4 = load i64, i64* %index, align 8
%5 = bitcast i8* %3 to i8**
call void @spactre_check_tag(i8** %5, i64 %4)
%arrayidx = getelementptr inbounds i8, i8* %3, i64 %4
%6 = load i8, i8* %arrayidx, align 1
```

Figure 4.11: LLVM IR of sPACTRE instrumentation of loads.

4.6 Implementation

The prototype for sPACTRE is built on top of Apple's LLVM fork [2] and runs on Apple silicon [28, 29, 31]. It consists of 2541 lines of code (LoC), with 2315 LoC for the LLVM pass and 226 LoC for the sPACTRE runtime library. The runtime library is integrated into

LLVM's compiler-rt.

Some optimizations are appied to SPACTRE. Link Time Optimization (LTO) is applied, which combines all the object files into one file. In addition to that, all the runtime library functions are inlined. The pass is executed in the LTO phase of optimizations, and this allows SPACTRE to take advantage of unnecessary instrumentation that would have been optimized out. Apple's PAC-based return address protection scheme is enabled.

4.7 Evaluation

SPACTRE is evaluated by answering the following questions:

- How effective is SPACTRE in preventing the malicious speculative access to a secret in the Spectre gadget? (§ 4.7.1)
- How much performance overhead does SPACTRE impose on synthetic and real-world cryptographic programs? (Section 4.7.2)

4.7.1 Security Evaluation

This section evaluates how SPACTRE prevents the speculative access to a secret. Prior work is relied on that has Proof of Concept (PoC) Spectre attacks implemented and tested on Apple Silicon [90]. This prior work implements the Spectre V1 and Spectre V2 attacks. A Spectre-RSB PoC is implemented based on SpectreRSB [105]. All the attacks on an Apple M1 and M3 are tested and verified that they succeed. Only one annotation to the secret variable is added, and compiled the programs with the SPACTRE compiler. SPACTRE is able to successfully defend against the attacks. Code snippets for the Spectre V1 (Section 4.7.1), Spectre V2 (Section 4.7.1), and Spectre-RSB (Section 4.7.1) PoCs are shown

```
// C-code
char* secret;
if(x < array1_size) {
    y = array1[x]; //accessor
    z = array2[y];} //transmitter
// Assembly
    ldr     x8, [sp, #8]
    ldrsw   x9, [sp, #28]
    ldr     w8, [x8, x9, lsl #2] // accessor load
    str     w8, [sp, #20]
    ldr     x8, [sp]
    ldrsw   x9, [sp, #20]
    ldr     w8, [x8, x9, lsl #2] // transmitter load
    str     w8, [sp, #16]
```

Figure 4.12: Spectre V1 code

```
// Instrumented LLVM IR
if.then:
  %7 = load i8*, i8** array2, align 8
  %auth.pi5 = ptrtoint i8* %7 to i64
  %8 = call i64 llvm.ptrauth.auth(i64 %auth.pi5, i32 2, ...
  %9 = inttoptr i64 %8 to i8*
  %10 = load i8*, i8** array1, align 8
  %auth.pi6 = ptrtoint i8* %10 to i64
  %11 = call i64 llvm.ptrauth.auth(i64 %auth.pi6, i32 2, ...
  %12 = inttoptr i64 %11 to i8*
  %13 = load i64, i64* %index, align 8
  %14 = bitcast i8* %12 to i8**
  call void spactre_check_tag(i8** %14, i64 %13)
  %arrayidx = getelementptr inbounds i8, i8* %12, i64 %13
  %15 = load i8, i8* %arrayidx, align 1
  %conv2 = sext i8 %15 to i32
  %idx = sext i32 %conv2 to i64
  %17 = bitcast i8* %9 to i8**
  call void spactre_check_tag(i8** %17, i64 %idxprom)
  %arrayidx4 = getelementptr inbounds i8, i8* %9, i64 %idx
  %18 = load i8, i8* %arrayidx4, align 1
```

Figure 4.13: LLVM IR snippet for the Spectre V1 PoC instrumented with sPACtre.

and how sPACtre defends against them. Note that for the code figures, both the C code and assembly are not instrumented with sPACtre compiler, and is only shown where the annotation would be and how it would like. The snippets shown below are simplified, and the full code is available in the prior work [90] and the sPACtre repo linked in Section 4.6.

Spectre V1 PoC

Figure 4.12 shows snippets of the C code and ARM assembly of the Spectre V1 PoC. The full code is shown in Figure 4.6. The victim function is used to train and poison the branch predictor to mispredict the conditional branch and speculatively access secret at an offset from array1. array1 contains some public data, and array2 is used to encode values into the cache state.

SPACTRE is able to defend against Spectre V1 by relying on the tag checking mechanism. The tag checking mechanism instruments spactre_check_tag to check tags accessed by array1 and array2 in Lines 4 and 5 respectively. The violation is detected speculatively when array1[x] in Line 4 goes out of bounds. When the check fails, the top two bits of the address are flipped and so the load in Line 13 will not result in a speculative leakage.

Defending against Spectre1.1 Spectre1.1 [102] is a Spectre attack that bypasses the bounds check in the conditional by relying on a store. It executes a ROP-style attack to bypass a barrier that may be placed at the conditional. Figure 4.14 shows a code example. c[y] points to the return address on the stack, and z contains the address of Line 3. When a return is done speculatively, execution will be redirected to the transmission gadget, bypassing a barrier that could have been added. SPACTRE is able to defend against this due to the existence of the PAC on the array pointer. The PAC needs to be authenticated before the load can be used. It is assumed here that the attacker would be able to bypass the metadata tag check instrumentation, however, they cannot bypass the authentication instrumentation, since the pointer wouldn't be usable speculatively without authenticating the PAC first. Thus, SPACTRE is able to defend against this attack.

```
1 int bar() {
2     if (x < lenb)
3         return a[b[x] * 512];
4 }
5 void bar(){
6     // z contains address to Line 4
7     if (y < lenc)
8         c[y] = z;}
```

Figure 4.14: Spectre1.1 code

Spectre V2 PoC

Figure 4.15 shows snippets of the C code and ARM assembly for the Spectre V2 PoC. The full code is shown in Figure 4.7. The branch target predictor is mistrained by calling secret_fun. Then, when executing the attack, dummy_fun is called. The function pointer attack is used to achieve this and is passed to the victim function. The mistrained branch at Line 14 gets executed speculatively, causing secret_fun to be executed speculatively, and leak the secret.

sPACtre defends against Spectre V2 due to the barrier instruction being inserted at the secret load due to the annotation of secret. After the mistraining of the branch target predictor, secret_fun is executed speculatively. The array access at Line 4 is where the secret would be leaked. However, due to the fact that the variable secret is annotated, this load has a barrier instruction inserted before it, and thus will not be executed speculatively, effectively blocking the attack.

All loads in the program have PACs on them, and the secret loads have PACs on them with a different modifier to the other loads. The barrier instruction blocks the speculation, but using the PAC with a unique modifier for secret helps in hardening the defense, and prevents the attacker from abusing other loads to leak the secret.

```
char* secret;
// Call this function on training.
void secret_fun(int x){
        y = secret[x];
        z = array2[y];}
// Call this function on misprediction.
void dummy_fun(int x){
        y = dummy[x];
        z = array2[y];}
void victim(void (**func)(int), size_t offset) {
    asm volatile(
      "MOV x0, %[off]\n" // offset
      "LDR x1, [%[func_ptr]]\n" // func
      "BLR x1\n" // mistrained branch
    :
    : [off] "r" (offset), [func_ptr] "p" (func)
    : "x0", "x1", "x30");}
```

Figure 4.15: Spectre V2 code

```
// Instrumented LLVM IR
define void secret_fun(i32 noundef %offset) #0 {
entry:
  ...
  %5 = load i8*, i8** secret, align 8
  %auth.pi4 = ptrtoint i8* %5 to i64
  %6 = call i64 @llvm.ptrauth.auth(i64 %auth.pi4, i32 2, ...
  %7 = inttoptr i64 %6 to i8*
  call void asm "DMB SY\0AISB SY", ""()
  %8 = load i64, i64* %index, align 8
  %9 = bitcast i8* %7 to i8**
  call void @spactre_check_tag(i8** %9, i64 %8)
  %arrayidx = getelementptr inbounds i8, i8* %7, i64 %8
  %10 = load i8, i8* %arrayidx, align 1
  ...}
```

Figure 4.16: LLVM IR snippet for the Spectre V2 PoC instrumented with SPACTRE.

Spectre-RSB PoC

Figure 4.17 shows a simplified pseudocode of the C code and ARM assembly for the Spectre-RSB PoC. The full code is shown in Figure 4.8. This is inspired by the SpectreRSB attack [105]. The gadget function gets called from main. gadget saves the return address (x30) to a return address array (return_addr_arr). It then calls the function thrash_cache, which

randomly accesses a huge array many times to thrash the cache. sPACTRE cannot use flush on ARM, since the flush instruction on ARM is a privileged instruction, unlike clflush in x86. gadget then calls the function called_function, which prepares the spectre_gadget that will be used to leak spectre_secret. called_function then replaces the return address in the Link Register x30 with the return address from return_addr_arr. This means that called_function does not return to gadget and returns to main instead. Thus, the secret in Line 19 would not be leaked architecturally. However, the RSB still has the original return address, and so the code in Lines 15-20 gets executed speculatively. This mismatch between the return address on the stack and the return address in the RSB causes the misspeculation and the secret to leak.

sPACTRE is able to defend against the Spectre-RSB attack due to the annotation of the secret (spectre_secret at Line 1 in Figure 4.17. This means that the load from spectre_secret in Line 19 will not be executed speculatively and thus the attack is prevented. Similarly to Spectre V2, the existence of the PAC on all loads, and the existence of PACs with unique modifiers to secret values helps to harden the defense against the Spectre-RSB attack.

Another RSB attack is ret2spec [121], which relies on RSB filling to overflow the RSB. One thing to note is that some ARM processors are vulnerable to the ret2spec, however, the Apple silicon chips are not vulnerable to ret2spec, due to the fact that the RSB gets cleared on overflow [16]. This is unlike AMD and Intel, in which the RSB can overflow, and the ret2spec attack can be executed.

Instrumented LLVM IR

To showcase how sPACTRE's instrumentation looks like, the instrumentation of the LLVM IR is shown for the snippets of the Spectre V1 and Spectre V2 PoCs. The defense for Spectre-

RSB is similar to Spectre V1. The authentication intrinsics are shown, spactre_check_tag instrumentation and the barrier instrumentation. Figure 4.13 and Figure 4.16 show a snippet of the LLVM IR Spectre V1 and Spectre V2 PoCs, respectively, instrumented with SPACTRE. The instrumented assembly is shown in Figure 4.9 and Figure 4.10 in § 4.5.2.

For the Spectre V1 instrumentation, the pointer load of array2 in Line 3 is used to access array2 in Line 20. This is checked with the pointer authentication intrinsics in Line 5. First, the pointer is converted to an integer with ptrtoint (Line 4). Then, the auth intrinsic is called, taking the converted pointer, secret key and modifier (Line 5). Then, the authenticated integer is converted back to a pointer with inttoptr. SPACTRE then finds all the uses of the original load and replaces it with the authenticated pointer, such as the array access in Line 20. In this way, the authenticated pointer is used, not the one with the PAC on it. spactre_check_tag is instrumented whenever there is a GEP instruction that accesses an array index (Line 14 and Line 20). In this way, any out-of-bounds access can be prevented.

For the Spectre V2 instrumentation, the pointer load of secret in Line 5 is used to access the secret in Line 14. This is also protected with the pointer authentication intrinsics (Line 7) and the metadata check (Line 12). However, due to the fact that this is a secret load, a barrier is also inserted in Line 9. In this way, the secret load cannot be abused by the attacker.

4.7.2 Performance Evaluation

Evaluation environment.

All the evaluations were run on a MacBook Pro with an Apple M3 processor [31], which supports the ARMv8.5 architecture. Pointer Authentication is in the ARMv8.3 Instruction Set Architecture (ISA), and therefore the Apple M3 supports hardware Pointer Authentication

```
char* spectre_secret;
static uint8_t spectre_buffer[256 * 256];
void* return_addr_arr[2048];
// Function that returns to address pointed
// to by argument instead of return address
void called_function() {
    x1 = &spectre_buffer[256];
    x30 = return_addr_arr[1024];
    return;}
void gadget(uint64_t idx){
    return_addr_arr[1024] = x30;
    thrash_cache(); // Randomly access large array
    called_function();
   // called_function does not return here.
   while(1) {
      // Executed speculatively.
      // Encode secret into cache.
      // Only executed during misprediction
      *(spectre_buffer + spectre_secret[idx] * 256);
   }
}
```

Figure 4.17: Spectre-RSB code

instructions. The sPACTRE prototype was implemented on Apple's fork of LLVM [2]. All the applications are with O2 for fair comparison.

Benchmarking crypto libraries

For the performance evaluation, sPACTRE relied on a variety of CT cryptographic libraries to showcase sPACTRE's performance, namely: libhydrogen [9], libsodium [14] and OpenSSL [17]. Libhydrogen has one cryptographic primitve that sPACTRE evaluate, Curve25519[42]. sPACTRE evaluates AES-256-GCM and ChaCha20 on Libsodium. Finally, sPACTRE evaluates AES-256-GCM, AES-256-OCM and ChaCha20 on OpenSSL. The libraries in a crypto benchmarking tool [7] were used, that were modified and patched. A message size of 64B and 8KB was used, and the benchmarking total runs 1,000,000 iterations on encrypt and decrypt.

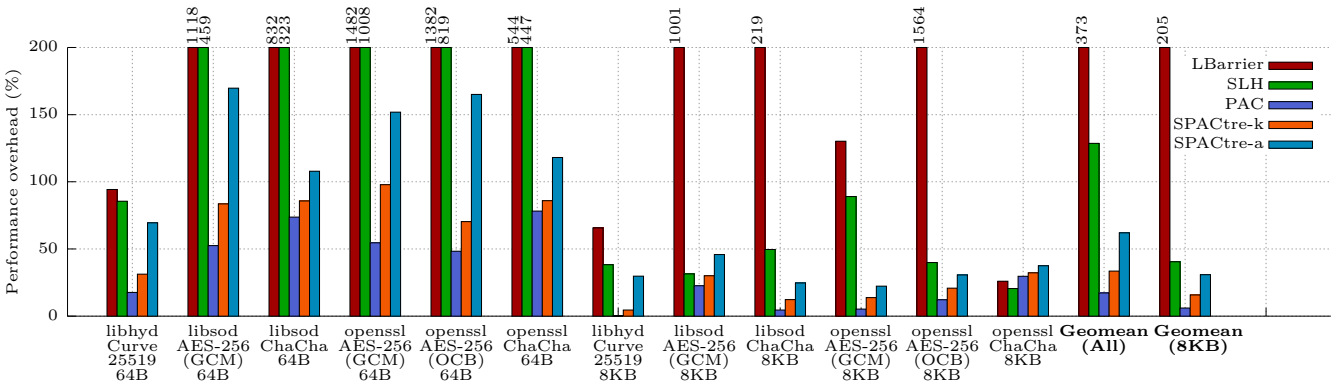


Figure 4.18: The performance overhead of different cryptographic algorithms and message sizes for libhydrogen, libsodium, openssl and the geometric mean for LoadBarrier, SLH, PAC only, SPACTRE-key and SPACTRE-all

In addition to this, SPACTRE is compared with two known Spectre mitigations. First, LoadBarrier was implemented, which places a barrier after every load. This has been used in prior work [133]. Second, the libraries with LLVM-SLH were enabled by passing the -mspeculative-load-hardening flag.

LoadBarrier and LLVM-SLH are compared against using the PAC on all loads/stores without the Spectre mitigation (PAC only), SPACTRE with only annotating the secret key and variables related to it (SPACTRE-key), and SPACTRE when annotating all variables related to the secret key and the message (SPACTRE-all).

Performance Overhead for crypto libraries

Figure 4.18 shows the performance overhead of SPACTRE. The larger message induces less overhead in general, and this is consistent with prior work [131]. The geometric mean in total for the benchmarks is 373%, 128.64%, 17.27%, 33.57% and 62.03% for LoadBarrier, LLVM-SLH, PAC-only, SPACTRE-key and SPACTRE-all, respectively. As can be seen, SPACTRE is able to significantly reduce the overhead compared to LoadBarrier and LLVM-SLH and provide more guarantees and protection. The overhead is even lower if only consider the

```
1  char* secret; // Secret data
2  void benchmark (int S, int C){
3          // Client workload with Spectre gadget
4          for (int i = 0; i < S; i++){
5                  work_load_with_spectre_gadgets();
6          }
7          // Encryption workload that uses a secret key
8          for (int i = 0; i < C; i++){
9                  encrypt_load();
10         }
11 }
```

Figure 4.19: Synthetic code pseudocode from SpectreGuard [80] composed of Spectre gadget and encryption workloads. S and C control the size of each respective workload.

geometric mean of only the 8KB message, with that being 6.07%, 15.78% and 30.85% for PAC-only, sPACtre-key and sPACtre-all, respectively.

Note that this also includes overhead of annotating all the loads and stores with PAC, and thus pointers are protected from being tampered with by memory corruption vulnerabilities in addition to the Spectre mitigation. The overhead added from the Spectre mitigation to a program with PAC only is around 16.3% and 51.3% for sPACtre-key and sPACtre-all, respectively, in total, and is 9.71% and 24.78% for sPACtre-key and sPACtre-all, respectively, if the message size is 8KB. Thus, sPACtre has relatively low overhead while providing strong security guarantees against Spectre control-flow attacks and memory corruption vulnerabilities.

Synthetic benchmark evaluation

sPACtre is evaluated with a synthetic benchmark from prior work, SpectreGuard [80]. Figure 4.19 shows the pseudocode of the benchmark. This benchmark mimics the runtime scenario of a sand-boxed application, such as a JavaScript engine in a browser, and is divided into two parts. The first part represents a client workload, which contains a Spectre gadget that is abused by the attacker. This can be the attacker's own JavaScript code. The

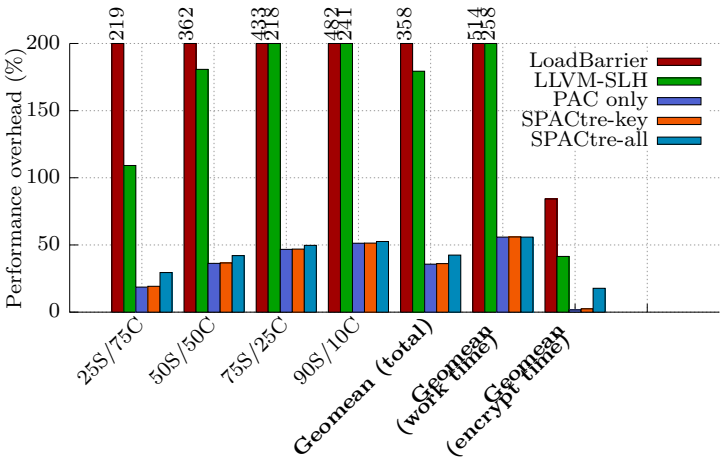


Figure 4.20: The performance overhead of the synthetic benchmark for different work and encryption times and the geometric mean for LoadBarrier, SLH, PAC only, SPACTRE-key, and SPACTRE-all

second part represents some background communication activities that a use a secret key for encryption, such as encrypted HTTPS communication. The benchmark uses AES encryption from OpenSSL. How much each part executes, denoted by 'S' for the client workload and 'C' for the encryption work load, is modified to simulate several scenarios. For example, "75S/25C" means that 75% of the time is on executing the client workload, and 50% of the time is on executing the encryption workload. LoadBarrier, LLVM-SLH, PAC only, SPACTRE-key, and SPACTRE-all are evaluated.

Figure 4.20 shows the results of the evaluation. The total geometric mean is 358%, 179.38%, 35.69%, 36.13%, 67.11% for LoadBarrier, LLVM-SLH, PAC-only, SPACTRE-key and SPACTRE-all. The geometric means of the client workload (work time) and the encryption time (encrypt time) are shown separately. The majority of the overhead is due to the work time. Also, SPACTRE-key has minimal overhead in the encrypt time, around 2.5%, since only the key is marked as secret. However, overhead increases to 17.78% for SPACTRE-all due to much more variables being marked as secret. Overall, SPACTRE introduces relatively low overhead in this synthetic scenario.

Summary

Across all evaluations, sPACtre offers much lower overhead than naively adding barriers to all loads in a program. Also, sPACtre has lower overhead than SLH whilst covering more Spectre variants. The performance results showcase the practicality of sPACtre in various scenarios.

4.8 Discussion and Limitations

4.8.1 Annotating the CT crypto libraries

The algorithms that need to be annotated are annotated manually in the cryptographic libraries that were used in evaluating sPACtre. The effort to annotate them was not significant. In general, it took less than 1 hour per algorithm to be able to annotate the secret variables and validate the correctness. Due to the nature of PAC changing pointer semantics, and the fact that the modifier changes when annotating secrets, it was relatively easy to confirm that the annotation was correct and covered all the variables needed to annotate the secret properly. This is attributed to the nature in which CT programs are written, in which careful consideration is already taken to ensure that secrets are not supplied to transmitters.

4.8.2 Deployability of sPACtre

Compared to other Spectre defense mechanisms [60, 183], sPACtre is deployable and can run on existing commercial hardware whilst providing strong security guarantees against Spectre attacks. More specifically, it can run on hardware that uses the ARM architecture

and supports Pointer Authentication, such as the Apple Silicon [28, 31, 173] processors. In addition to that, SPACTRE is a defense mechanism for the ARM architecture, which is under-explored in terms of Spectre defense mechanisms. Note that, even though SPACTRE is currently implemented and specific to ARM, it can be modified and ported to other architectures if a hardware security feature such as Pointer Authentication is made available.

PAC is not only available on the ARM architecture. Mechanisms similar to PAC exist in software and hardware. Cryptographic Control Flow Integrity (CCFI) uses Message Authentication Codes (MACs) to protect return addresses, function pointers and VTable pointers by generating a MAC and placing it on the top bits of a pointer. CCFI is a software implementation. Efforts have been made to deploy PAC on RISC-V [19]. SPARC's implementation of a similar mechanism is called Application Data Integrity (ADI) [1]. Thus, SPACTRE can be extended to work on other architectures as well if the compatible hardware security primitives are available.

4.8.3 Manual annotation effort

A limitation of SPACTRE is its need for programmer annotation of the secrets in a program. This means that SPACTRE needs the program to have the secret clearly defined so that it can be protected and for the programmer to explicitly annotate them. This allows SPACTRE to be a good fit for Constant Time (CT) programs, but may not be optimal for larger and more complex programs. Even though annotation efforts are needed, the trade-off in terms of efficiency and strong security guarantees is worth it.

4.8.4 Compatibility with memory corruption defense mechanisms

Defending against memory corruption attacks is orthogonal to defending against Spectre attacks. sPACTre's mechanism allows it to be deployed with other defense mechanisms. Thus, sPACTre can be deployed alongside memory corruption defenses [24, 49, 97, 106, 164] to protect against attacks that rely on memory corruption [46, 52, 147, 155].

4.9 Related Work

4.9.1 Software mitigations

Software mitigations against Spectre rely on pure software techniques to defend against different Spectre variants. Other than the ones discussed in Section 4.1, other mitigations are listed below.

oo7 [174] is a static analysis technique that detects potentially vulnerable code snippets in the program's binary and patches them to protect against Spectre attacks. Blade [172] relies on its own type system to classify each expression as either transient or stable. The main insight is to stop data flowing from expressions that can leak secrets. UltimateSLH [188] further strengthens SLH by providing protection against variable-time arithmetic instruction, that can steal leak secrets even with SLH enabled. Venkman [157] prevents Spectre-BTB and Spectre-RSB attacks by transforming the code so that all valid control-flow transfer targets have an identical alignment in the virtual address space, and further transforms all the branches to guarantee that all entries in the RSB and BTB are aligned properly.

With the exception of Swivel [133] and Serberus [131], all these techniques prevent one or two, but not all three Spectre control-flow variants. In addition to that, some have very

high overheads, such as Intel-lfence and LLVM-SLH. Swivel is specifically targeted towards Wasm and Serberus relies on Intel-CET to enforce its protection. None of these defenses, except LLVM-SLH, are implemented in the ARM architecture. Thus, SPACTRE provides an efficient and effective contribution to the Spectre defense mechanisms.

4.9.2 Implemented hardware mitigations

Some mitigations exist that are prevented in hardware by the hardware vendor. Intel enhanced Indirect Branch Restricted Speculation (IBRS) [10] is a hardware solution that manages and controls predictor modes to ensure that lower-privilege software cannot control indirect branch prediction. ARM CSV2 [8] is a similar solution on ARM to Intel's IBRS that intends to prevent cross-privilege branch attacks. Another solution to preventing these attacks is completely disabling the branch predictor.

Both IBRS and CSV2 only prevent cross-privilege attacks, and are still vulnerable if the mis-prediction is of the victim's own predictions [38]. SPACTRE is not vulnerable to this. It is not possible to disable the branch predictor on the Apple M3, due to the fact that the MRS/MSR instructions, that are used to read and write from and to special system registers, are only accessible from the kernel. However, it is expected that disabling the branch predictor would incur a high overhead. SPACTRE can still defend against Spectre-BTB attacks without disabling the branch predictor.

4.9.3 Unimplemented hardware mitigations

There are mitigation techniques that propose new hardware to defend against Spectre. These mainly rely on using a simulator or a RISC-V implementation. Speculative Taint Tracking (STT) [183] selects and forwards specific results of speculative execution if it can prove that

the forwarded results do not transmit to a covert side channel. Speculative Privacy Tracking (SPT) [60] delays the execution of a transmitter until it can prove that its operands leak during non-speculative execution, and thus it is safe. Both STT and SPT have simulator implementations and thus aren't deployable on existing hardware. sPACtre is deployable on existing hardware.

4.9.4 Annotation mitigations

There have been Spectre mitigations that rely on annotations as well. ConTExT [153] relies on allowing secrets to enter registers but not transiently leave them. It relies on adding a new non-transient bit to the page-table entry. If enabled, a dummy value is used instead of the secret value when transient execution occurs. ProSpeCT [66] tracks secrets in the processor pipeline and ensures that they do not change the microarchitectural state when speculative execution happens. ProSpeCT relies on annotating secrets in Constant Time (CT) programs.

Both ConTExT and ProSpeCT show that there is value in annotating secrets in programs to defend against Spectre attacks, and both showcase low performance overhead in their respective evaluations, similar to sPACtre. However, ConTExT is implemented by simulation and ProSpeCT is implemented as a RISC-V implementation. Thus, they are not deployable, whereas sPACtre uses existing hardware features coupled with software mitigations and is deployable on existing hardware.

4.10 Summary

This dissertation introduced SPACTRE, a new and novel defense mechanism that aims to prevent the malicious speculative access of a secret in the Spectre gadget. SPACTRE is an ARM-based defense mechanism relies on ARM's Pointer Authentication hardware security feature to harden the code, annotations added to the program on the secrets that need to be protected from leakage and a dynamic tag checking mechanism for arrays. This dissertation showed that SPACTRE can defend against Spectre attacks, and showcased SPACTRE's defense against the Spectre-PHT, Spectre-BTB and Spectre-RSB attacks. This dissertation evaluated SPACTRE on a variety of cryptographic libraries with several cryptographic algorithms, as well as a synthetic benchmark, and showed that is efficient and has low performance overhead.

Chapter 5

Future research direction

This chapter of the dissertation outlines possible future work to build on top of the works already discussed. For the future work, this dissertation proposes extending protection with hardware security features to cover energy harvesting devices. Energy harvesting devices are usually bare-metal devices with little or no defense mechanisms against system attacks. In addition to that, the power failures that happen may end up corrupting critical information. Section 5.2 gives a brief background on energy harvesting devices and the possible system attacks and corruptions against them. Section 5.3 gives a brief idea on what the limitations are of current defense schemes and what a future defense would aim to solve.

5.1 Energy harvesting systems

Energy harvesting devices are batteryless, and store the harvested energy in their capacitors to power all their hardware components. When these capacitors run out of energy, power failures happen. This interrupts the devices running computing and could cause the device to lose some of its results and its volatile state. When the device is restarted, the program

restarts again, which slows down the progress of the program. In addition to that, executing blocks of code again could lead to issues with data consistency.

In order to recover the data properly, several approaches have been proposed. State-of-the-art approaches are classified either into task-based systems or checkpointing systems.

For task-based systems [61, 89, 120, 122, 180], the developer writes programs that adhere to a certain programming model. The system then reduces the computation to a group of tasks during compilation. Then, it determines the control-flow and manages the data that the tasks manipulate and share. These tasks are atomic in nature, and thus they must entirely execute. This is done so that there are no inconsistencies in non-volatile memory in case of a power failure. Thus, these task-based mechanisms guarantee that tasks complete even if power failures occur. However, the process of dividing the computation to a group of tasks, determining the control flow and managing the shared data requires a great amount of effort from the developer, and is susceptible to errors [104].

In comparison to task-based systems, checkpointing systems are easier to develop for, since they only need either automatic or manual checkpoints to be added to C programs [37, 104, 119, 146, 177]. The main function of checkpoints is to preserve the current state of the program in the FRAM before the power fails. After the device is restarted, it restores the state of the program using the last stored successful checkpoint from the FRAM.

5.2 Attacks against energy harvesting systems and current defenses

These embedded devices are exposed to more attacks, due to the impacting results that occur from the constant power failures. This causes them to be vulnerable to attacks that

transfer power and others [99, 119, 145, 148].

Systems on energy harvesting devices can be attacked through the network by means of attacks such as:

- **Eavesdropping:** Energy harvesting networks are implemented usually without a built-in encryption mechanism. This exposes the information and energy to eavesdropping attacks.
- **Denial-of-Service(DoS):** One of the ways to exhaust the limited energy available on energy harvesting devices are what is referred to as energy depletion attacks. Energy depletion attacks are a type of DoS attack, where the attacker sends fake packets to nodes that are wirelessly powered.
- **Malware infection:** Another way to deplete the energy harvesting device's energy is with a malicious application. This can send several requests for energy, thus degrading the throughput of the system.

These types of attacks compromise the device and its ability to function and recover from failures. Several defenses have been proposed. They have relied on authenticity-based integrity checking [162] or defining a new architecture for secure energy harvesting batteryless devices [71]. However, none of these approaches guarantee the security of checkpoints from an adversary that can carry out remote software attacks.

5.3 Possible future defense approach

A promising approach is to rely on the use of cryptographic-based integrity checking of stored checkpoints [162]. This approach utilizes cryptographic primitives to compute a MAC that would verify that the stored checkpoints are fresh and authentic. It relies on a 128 bit nonce and a 128 bit tag that are stored in tamper-free memory, which is new hardware that is added.

The nonce and tag are used to generate an authentication tag by relying on a Message Authentication Code (MAC) from a hash-based cryptographic algorithm. However, this approach has relatively high performance overhead and does not take into account an attacker that can execute memory corruption vulnerabilities. Moreover, this defense mechanism relies on new custom hardware, the tamper-free memory, and thus is not deployable.

A possible future direction is the use of a hardware based cryptographic primitive that would not need the tamper-free memory free, similar to PAC [95]. ARM has released PACBTI [18] in ARMv8.1-M. PACBTI is inspired by the Pointer Authentication feature released ARMv8.3 [95] and Branch Target Identification released in ARMv8.5 [5]. However, this is optimized for embedded systems security. An interesting direction would be exploring PACBTI to see if it can efficiently and effectively protect checkpoints in commodity systems without relying on tamper-free memory. It would possibly have better overhead due to the execution in hardware. There aren't any energy harvesting devices currently that support PACBTI, but this could change in the future as it becomes more widely implemented, similar to what happened with PAC when it got adopted into Apple silicon [30].

Chapter 6

Conclusion

This dissertation presented practical mitigations against control-flow hijacking, data-oriented and speculative execution attacks by leveraging hardware security features, such as ARM's Pointer authentication.

PACTight is an efficient and robust mitigation to guarantee pointer integrity using ARM's Pointer Authentication mechanism to mitigate control-flow hijacking attacks. PACTight enforces three security properties to ensure pointer integrity: (1) Unforgeability, (2) Non-copyability, (3) Non-dangling. PACtight was implememented with four defense mechanisms, protecting forward edge, backward edge, virtual function pointers, and sensitive pointers.

Scope-Type Integrity (STI) is 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. This dissertation presented RSTI, a robust and efficient security mechanism that protects all pointers in a program by leveraging ARM Pointer Authentication.

sPACtre is a new and novel defense mechanism that aims to prevent Spectre control-flow

attacks on existing hardware. sPACtre is an ARM-based defense mechanism that prevents Spectre attacks by relying on ARM's Pointer Authentication hardware security feature, annotations added to the program on the secrets that need to be protected from leakage and a dynamic tag-based bounds checking mechanism for arrays.

Finally, this dissertation introduced a new direction for utilizing hardware security features to protect energy harvesting devices from checkpoint-recovery errors and malicious attackers.

These three mechanisms and the future direction showcase that practical mitigations against memory corruption and transient execution attacks are possible.

Bibliography

[1] Application Data Integrity on SPARC. hhttps://www.kernel.org/doc/html/v5.7/sparc/adi.html.

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

[3] DMB and DSB and ISB, . https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/miscellaneous-instructions/dmb--dsb--and-isb.

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

[5] ARM Branch Target Identification. https://developer.arm.com/documentation/ddi0602/latest/Base-Instructions/BTI--Branch-Target-Identification.

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

[7] Mumble's crypto benchmark. https://github.com/mumble-voip/crypto-benchmark.

[8] ARM CSV2. https://documentation-service.arm.com/static/623c60d13b9f553dde8fd8e6?token=.

[9] libhydrogen. https://github.com/jedisct1/libhydrogen.

[10] Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[11] Using Intel® Compilers to Mitigate Speculative Execution Side-Channel Issues. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues.html.

[12] Load Fence (lfence). https://www.felixcloutier.com/x86/lfence.

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

[14] libsodium, . https://github.com/jedisct1/libsodium.

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

[16] Apple M1 return address prediction stacks. https://twitter.com/dougallj/status/1580826539205496832.

[17] OpenSSL: Cryptography and SSL/TLS toolkit. https://www.openssl.org/.

[18] Armv8.1-M Pointer Authentication and Branch Target Identification Extension., . https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension.

[19] RISC-V ISA Extension for Control Flow Integrity, . https://www.sec.in.tum.de/i20/student-work/risc-v-isa-extension-for-control-flow-integrity.

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

[21] Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[22] Speculative Load Hardening (SLH). https://llvm.org/docs/SpeculativeLoadHardening.html.

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

[24] 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.

[25] 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.

[26] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

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

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

[29] 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.

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

[31] Apple. Apple unveils M3, M3 Pro, and M3 Max, 2023. https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/.

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

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

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

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

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

[37] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12): 1968–1980, 2016. doi: 10.1109/TCAD.2016.2547919.

[38] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX*

Security 22), pages 971–988, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[39] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021. doi: 10.1109/SP40001.2021.00008.

[40] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. URL https://doi.org/10.1145/3371075.

[41] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *Proceedings of the 12th International Conference on Fast Software Encryption*, FSE'05, page 32–49, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540265414. URL https://doi.org/10.1007/11502760_3.

[42] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *International Conference on Theory and Practice of Public Key Cryptography*, 2006. URL https://api.semanticscholar.org/CorpusID:6720831.

[43] 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, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. URL https://doi.org/10.1145/3319535.3363194.

[44] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative exploitation of ROP chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 1–16, San Sebastian, October 2020. USENIX Association. ISBN 978-1-939133-18-2. URL https://www.usenix.org/conference/raid2020/presentation/bhattacharyya.

[45] 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.

[46] 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.

[47] 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.

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[49] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brun-

thaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76, 2017. doi: 10.1109/SecDev.2017.24.

[57] 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.

[58] 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.

[59] 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.

[60] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 607–622, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480068. URL https://doi.org/10.1145/3466752.3480068.

[61] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*,

OOPSLA 2016, page 514–530, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344449. doi: 10.1145/2983990.2983995. URL https://doi.org/10.1145/2983990.2983995.

[62] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009. doi: 10.1109/SP.2009.19.

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

[64] 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.

[65] 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.

[66] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. ProSpeCT: Provably secure speculation for the Constant-Time policy. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7161–7178, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/daniel.

[67] 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.

[68] 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.

[69] Sushant Dinesh, Grant Garrett-Grossman, and Christopher W. Fletcher. Synthct: Towards portable constant-time code. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL https://www.ndss-symposium.org/ndss-paper/auto-draft-223/.

[70] 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.

[71] Daniel Dinu, Archanaa S. Khrishnan, and Patrick Schaumont. Sia: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 208–217, 2019. doi: 10.1109/HST.2019.8740834.

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

[73] 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.

[74] 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.

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

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

[77] 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.

[78] 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.

[79] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using intel sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 765–782, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. URL https://doi.org/10.1145/3133956.3134106.

[80] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[81] 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.

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

[83] 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.

[84] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[85] Jens Grossklags and Claudia Eckert. τCFI: 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.

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

[87] 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.

[88] 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.

[89] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450354592. doi: 10.1145/3131672.3131673. URL https://doi.org/10.1145/3131672.3131673.

[90] Lorenz Hetterich and Michael Schwarz. Branch different - spectre attacks on apple silicon. In *International Conference on Detection of intrusions and malware, and vulnerability assessment*, 2022. URL https://api.semanticscholar.org/CorpusID:250180821.

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

[92] Jann Horn. Speculative Execution, Variant 4: Speculative Store Bypass., 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[93] 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.

[94] 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.

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

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

[97] 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.

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

[99] Jiawen Kang, Rong Yu, Sabita Maharjan, Yan Zhang, Xumin Huang, Shengli Xie, Hanna Bogucka, and Stein Gjessing. Toward secure energy harvesting cooperative networks. *IEEE Communications Magazine*, 53(8):114–121, 2015. doi: 10.1109/MCOM.2015.7180517.

[100] 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.

[101] 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.

[102] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *ArXiv*, abs/1807.03757, 2018. URL https://api.semanticscholar.org/CorpusID:49666976.

[103] 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.

[104] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 85–99, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378476. URL https://doi.org/10.1145/3373376.3378476.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authen-

tication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 177–194, Santa Clara, CA, August 2019.

[112] 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.

[113] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[114] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[115] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[116] 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.

[117] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015. doi: 10.1109/SP.2015.43.

[118] 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.

[119] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 575–585, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737978. URL https://doi.org/10.1145/2737924.2737978.

[120] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133920. URL https://doi.org/10.1145/3133920.

[121] 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, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. URL https://doi.org/10.1145/3243734.3243761.

[122] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sen. Netw.*, 16(1), feb 2020. ISSN 1550-4859. doi: 10.1145/3360285. URL https://doi.org/10.1145/3360285.

[123] 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.

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

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

[126] Michael Larabel. Glibc Rolls Out Support For Memory Protection Keys, 2017. https://www.phoronix.com/scan.php?page=news_item&px=Glibc-Memory-Protection-Keys.

[127] microsoft. Mitigating speculative execution side channel hardware vulnerabilities, 2018. https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[128] 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.

[129] 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.

[130] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program

counter security model: Automatic detection and removal of Control-Flow side channel attacks. Baltimore, MD, July 2005. USENIX Association.

[131] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel. Serberus: Protecting cryptographic code from spectres at compile-time. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 47–47, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. doi: 10.1109/SP54263.2024.00048. URL https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00048.

[132] 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.

[133] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

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

[135] 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.

[136] 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.

[137] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

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

[139] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32648-9.

[140] 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.

[141] Colin Percival. Cache missing for fun and profit. 2005. URL https://api.semanticscholar.org/CorpusID:14403674.

[142] 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.

[143] 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.

[144] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security*

21), pages 1451–1468. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/ragab.

[145] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 221–236, Bellevue, WA, August 2012. USENIX Association. ISBN 978-931971-95-9. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati.

[146] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: system support for long-running computation on rfid-scale devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 159–170, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450302661. doi: 10.1145/1950365.1950386. URL https://doi.org/10.1145/1950365.1950386.

[147] 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, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527429. URL https://doi.org/10.1145/3470496.3527429.

[148] Arman Roohi, Ronald F DeMara, Longfei Wang, and Selçuk Köse. Secure intermittent-robust computation for energy harvesting device security and outage resilience. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, In-*

ternet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), pages 1–5, 2017. doi: 10.1109/UIC-ATC.2017.8397617.

[149] 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.

[150] 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.

[151] 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.

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

[153] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *Network and Distributed System Security Symposium*, 2020. URL https://api.semanticscholar.org/CorpusID:211268150.

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

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

[156] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1211–1228, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. URL https://doi.org/10.1145/3133956.3134095.

[157] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. *ArXiv*, abs/1903.10651, 2019. URL https://api.semanticscholar.org/CorpusID:85517893.

[158] 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.

[159] 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.

[160] 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.

[161] 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.

[162] Charles Suslowicz, Archanaa S. Krishnan, Daniel Dinu, and Patrick Schaumont. Secure application continuity in intermittent systems. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2018. doi: 10.1109/IGCC.2018.8752145.

[163] 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.

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

[165] 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.

[166] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434330. URL https://doi.org/10.1145/3434330.

[173] 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.

[174] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 47:2504–2519, 2018. URL https://api.semanticscholar.org/CorpusID:208303629.

[175] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. URL https://doi.org/10.1145/3290390.

[176] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/wikner.

[177] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude.

[178] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3), may 2021. ISSN 0360-0300. URL https://doi.org/10.1145/3442479.

[179] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX*

Security 14), pages 719–732, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[180] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359528. doi: 10.1145/3274783.3274837. URL https://doi.org/10.1145/3274783.3274837.

[181] 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.

[182] 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.

[183] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358274. URL https://doi.org/10.1145/3352460.3358274.

[184] 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.

[185] 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.

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

[187] 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.

[188] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh.

[189] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1789–1806, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. URL https://doi.org/10.1145/3133956.3134043.