PACMAN: Attacking ARM Pointer Authentication with Speculative Execution

Joseph Ravichandran*
MIT CSAIL
Cambridge, MA, USA
jravi@mit.edu

Weon Taek Na*
MIT CSAIL
Cambridge, MA, USA
weontaek@mit.edu

Jay Lang
MIT CSAIL
Cambridge, MA, USA
jaytlang@mit.edu

Mengjia Yan
MIT CSAIL
Cambridge, MA, USA
mengjiay@mit.edu

ABSTRACT

This paper studies the synergies between memory corruption vulnerabilities and speculative execution vulnerabilities. We leverage speculative execution attacks to bypass an important memory protection mechanism, ARM Pointer Authentication, a security feature that is used to enforce pointer integrity. We present PACMAN, a novel attack methodology that speculatively leaks PAC verification results via micro-architectural side channels without causing any crashes. Our attack removes the primary barrier to conducting control-flow hijacking attacks on a platform protected using Pointer Authentication.

We demonstrate multiple proof-of-concept attacks of PACMAN on the Apple M1 SoC, the first desktop processor that supports ARM Pointer Authentication. We reverse engineer the TLB hierarchy on the Apple M1 SoC and expand micro-architectural side-channel attacks to Apple processors. Moreover, we show that the PACMAN attack works across privilege levels, meaning that we can attack the operating system kernel as an unprivileged user in userspace.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Hardware reverse engineering**.

KEYWORDS

Security, memory corruption attacks, micro-architectural side channels, pointer authentication

ACM Reference Format:
Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527429

*Both authors contributed equally to this research.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527429

1 INTRODUCTION

Modern systems are becoming increasingly complex, exposing a large attack surface with vulnerabilities in both software and hardware. In the software layer, memory corruption vulnerabilities [16, 56, 59, 61] (such as buffer overflows) can be exploited by attackers to alter the behavior or take full control of a victim program. In the hardware layer, micro-architectural side channel vulnerabilities [18, 25] (such as Spectre [37] and Meltdown [43]) can be exploited to leak arbitrary data within the victim program's address space. Today, it is common for security researchers to explore software and hardware vulnerabilities separately, considering the two vulnerabilities in two disjoint threat models.

In this paper, we study the synergies between memory corruption vulnerabilities and micro-architectural side-channel vulnerabilities. We show how a hardware attack can be used to assist a software attack to bypass a strong security defense mechanism. Specifically, we demonstrate that by leveraging speculative execution attacks, an attacker can bypass an important software security primitive called *ARM Pointer Authentication* [55] to conduct a control-flow hijacking attack.

ARM Pointer Authentication. Memory corruption vulnerabilities [22, 61, 65] pose a significant security threat to modern systems. These vulnerabilities are caused by software bugs which allow an attacker to corrupt the content of a memory location. The corrupted memory content, containing important data structures such as code and data pointers, can then be used by the attacker to hijack the control flow of the victim program. Well-studied control-flow hijacking techniques include return-oriented programming (ROP) [56] and jump-oriented programming (JOP) [16].

In 2017, ARM introduced Pointer Authentication (PA for short) in ARMv8.3 [55] as a security feature to protect pointer integrity. Since 2018, Pointer Authentication has been supported in Apple processors, including multiple generations of mobile processors [33] and the recent M1, M1 Pro, and M1 Max chips [5, 7]. Multiple chip manufactures, including ARM, Qualcomm, and Samsung, have either announced or are expected to ship new processors supporting Pointer Authentication. In a nutshell, Pointer Authentication is currently being used to protect many systems, and is projected to be even more widely adopted in the upcoming years.

Pointer Authentication makes it significantly more difficult for an attacker to modify protected pointers in memory without being

detected. Pointer Authentication protects a pointer with a cryptographic hash. This hash verifies that the pointer has not been modified, and is called a *Pointer Authentication Code*, or *PAC* for short. Considering that the actual address space in a 64-bit architecture is usually less than 64 bits, e.g., 48 bits on macOS 12.2.1 on M1, PA stores the PAC together with the pointer in these unused bits. Whenever a protected pointer is used, the integrity of the pointer is verified by validating the PAC using the pointer value (more details are in Section 2.2). Use of a pointer with an incorrect PAC will cause the program to crash. With Pointer Authentication in place, an attacker who wants to modify a pointer must correctly guess or infer the matching PAC of the pointer after modification.

Depending on the system configuration, the size of the PAC, which ranges from 11 to 31 bits [55], may be small enough to be bruteforced. However, simple bruteforcing approaches cannot break PA. The reason is that every time an incorrect PAC is used, the event results in a victim program crash. Restarting a program after a crash results in changed PACs, as the PACs are computed from renewed secret keys. Moreover, frequent crashes can be easily captured by anomaly detection tools [26].

The PACMAN Attack. In this paper, we propose the PACMAN attack, which extends speculative execution attacks to bypass Pointer Authentication by constructing a *PAC oracle*. Given a pointer in a victim execution context, a PAC oracle can be used to precisely distinguish between a correct PAC and an incorrect one without causing any crashes. We further show that with such a PAC oracle, the attacker can brute-force the correct PAC value while suppressing crashes and construct a control-flow hijacking attack on a PA-enabled victim program or operating system.

The key insight of our PACMAN attack is to use speculative execution to stealthily leak PAC verification results via micro-architectural side channels. Our attack works relying on *PACMAN gadgets*. A PACMAN gadget consists of two operations: 1) a pointer verification operation that speculatively verifies the correctness of a guessed PAC, and 2) a transmission operation that speculatively transmits the verification result via a micro-architectural side channel. The pointer verification operation is performed by an authentication instruction (new instructions in ARMv8.3), which outputs a valid pointer if the verification succeeds and an invalid pointer otherwise. The transmission operation can be performed by a memory load/store instruction or a branch instruction taking the output pointer as an address. If a correct PAC is guessed, the transmission operation will speculatively access a valid pointer, resulting in observable micro-architectural side effects. Otherwise, the transmission step will cause a speculative exception due to accessing an invalid pointer. Note that we execute both operations on a mis-speculated path. Thus, the two operations will not trigger architecture-visible events, avoiding the issue where invalid guesses result in crashes.

We provide multiple proof-of-concept demonstrations of the PACMAN attack on the Apple M1 SoC, the first desktop processor that supports Pointer Authentication. We identified multiple key challenges involved in researching Apple platforms. To begin with, Apple platforms, including both the hardware and the operating system, are scarcely documented. To the best of our knowledge, there was no public documentation with enough micro-architectural details about Apple M1 processors for us to start a side-channel attack during the timeframe of this project. Additionally, we found the M1 SoC, by default, to not expose a high-resolution timer to userspace, which is generally essential for micro-architecture reverse engineering.

In this paper, in addition to proposing the novel PACMAN attack, we make significant contributions in expanding micro-architectural side-channel attacks to Apple processors. We highlight two important outcomes. First, we reverse engineer the TLB organizations and perform the *first* TLB-based side-channel attack with speculative execution on Apple M1 processors. Second, we show that the PACMAN attack works *across privilege levels*, implying the feasibility of attacking a PA-enabled operating system kernel.

Impact. Our attacks have significant security impact both in practice and academically. From a practical perspective, our attack is general enough to be applicable to future ARM processors. Pointer Authentication is becoming increasingly popular, and many chip manufactures are planning to ship processors supporting it in the near future [12]. If not mitigated, our attack will affect the majority of mobile devices, and likely even desktop devices in the coming years.

From an academic perspective, on one hand, we have seen an extensive list of work that extends PA [23, 36, 40, 41, 49], whose security properties have been examined solely under the memory safety threat model. Since our attack breaks Pointer Authentication, our work calls for re-evaluating the security properties of those extended designs under a broader threat model involving speculative execution attacks. On the other hand, fundamentally, our attack highlights that security mechanisms that employ a security-by-crash design principle and rely on low collision probability are potentially vulnerable to speculative execution attacks, since speculation can serve as a primitive to suppress crashes. We envision that our work can inspire future explorations of both attacks and defenses to address the synergies between memory corruption vulnerabilities and speculative execution vulnerabilities.

Contributions. The main contributions of this paper can be summarized as follows:

- We identify fundamental security limitations of ARM Pointer Authentication and propose the PACMAN attack to bypass PA without causing crashes (Section 4).
- We develop micro-architectural timing side-channel attack primitives and a reverse engineering tool for Apple M1 processors (Section 6).
- We reverse engineer the memory hierarchy of Apple M1 processors, for the first time revealing micro-architectural details of the TLBs on the Apple M1 SoC (Section 7).
- We demonstrate several proof-of-concept of PACMAN attacks, including constructing PAC oracles, brute-forcing PACs, and a control-flow hijacking attack targeting a PA-enabled kernel module (Section 8).

Open-source Release. We have open-sourced our tools and proof-of-concept attacks here: pacmanattack.com

2 BACKGROUND

2.1 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities are an old security problem [4]. Having existed for more than 30 years, they still continue to be a serious threat to modern systems [56, 61]. This type of vulnerability exists in software written in low-level languages, such as C and C++. According to the MITRE 2021 rankings [48], the top 10 most dangerous software weaknesses include multiple memory corruption bugs, such as out-of-bound writes, out-of-bound read, and use-after-free. Other memory corruption bugs include double-free bugs, integer overflows and underflows, size confusion attacks, and type confusion attacks.

A memory corruption attack exploits a software bug to corrupt the content of a memory location, which contains important data structures, such as data and code pointers. The set of attacks that modify code pointers to change the control flow of the victim program is called *control-flow hijacking attacks*, such as code injection attacks, return-oriented programming (ROP) [31, 56, 66], and jump-oriented programming (JOP) [11, 16].

Modern systems commonly adopt several forms of memory vulnerability protection mechanisms. These mitigations include stack canaries [19], data execution prevention [64], address space layout randomization (ASLR) [1, 47], and kernel address space layout randomization (kASLR) [21]. Even though these mechanisms make memory corruption attacks more difficult, several advanced attacks [11, 16, 31, 56, 66] and various data disclosure attacks [29, 59] still have the potential to bypass these memory protection mechanisms, and show that none of the existing systems are impenetrable.

2.2 ARM Pointer Authentication

ARM introduced *Pointer Authentication* [55] (ARM *PA* for short) in the ARMv8.3 instruction set [9] to protect pointer integrity and make it significantly more difficult for an attacker to modify a protected pointer in memory without being detected. This security feature provides cryptographically strong guarantees that pointers have not been tampered with by adversaries while optimizing performance and attaining low runtime overhead. To this end, PA adds two sets of instructions and a cryptography implementation in hardware to *sign* and *verify* pointers.

Given a pointer to protect, PA generates a cryptographic hash of the pointer. This hash is used as the signature of the pointer and is called a *Pointer Authentication Code* or *PAC*[1] for short. Since the actual address space in a 64-bit architecture is usually less than 64 bits, PA stores the PAC together with the pointer in these unused bits.

Signing and Verifying Pointers. We show the signing and verifying process of AM PA in Figure 1. To sign a pointer (Figure 1(a)), the processor computes the PAC from the pointer value and a context value, and stores the PAC in the upper unused bits in the pointer. The context value consists of a *key*, stored in a privileged register inaccessible to userspace, and a program-specified *salt*.

[1]It is common to refer to Pointer Authentication, the security feature, as PAC. In this paper, we use "PA" to represent the Pointer Authentication feature, and "PAC" to represent the cryptographic hash inserted into a pointer.

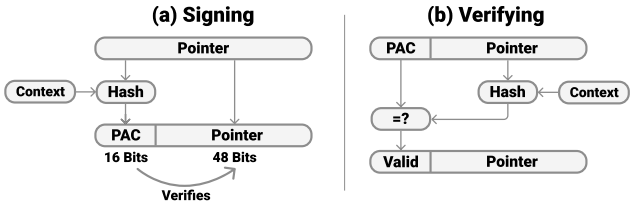


Figure 1: Signing and verifying pointers.

```
1 pacia lr, sp
2 sub   sp, sp, #0x40
3 str   lr, [sp, #0x30]
4 ...
```

(a) Function prologue.

```
1 ldr   lr, [sp, #0x30]
2 add   sp, sp, #0x40
3 autia lr, sp
4 ret
```

(b) Function epilogue.

Figure 2: Stack protection using ARM Pointer Authentication.

To verify a pointer (Figure 1(b)), i.e., to check whether a pointer has been tampered with or not, the processor re-computes the PAC from the pointer and the same context value, and compares the re-computed PAC against the PAC bits embedded in the passed pointer. If the two values match, the PAC bits will be cleared and the resulting pointer can be used normally. Otherwise, the processor will store an invalid pointer in the output register by setting up certain bits in the PAC region. De-referencing such a pointer will trigger a virtual address translation fault. Notably, PA is not transparent to the ISA; software must use special instructions to gain the benefits of PA.

ISA Extension. To accomplish the above functionality, PA introduces two sets of instructions. The `pac` prefix denotes instructions used for signing pointers, and the `aut` prefix denotes instructions for verifying pointers. Recall that a context value of computing a PAC includes a key and a salt, which are encoded differently. PA can store up to five keys at a time in hardware, and which key to use is encoded in the instruction's opcode. The salt is passed in as an operand. For instance, "`pacia ptr, salt`" signs an instruction pointer using the IA key, indicated by the `i` and the trailing `a` in the opcode.

We show an example of protecting return addresses using PA in Figure 2. We follow ARM's symbolic register naming convention, where `lr` stands for the procedure link register for storing the return address, and `sp` stands for the stack pointer. The first argument for each instruction is the destination register.

During the function prologue in Figure 2(a), the return address (`lr`) of the given function is signed using the stack pointer (`sp`) as a salt on line 1. The signed return address is then pushed onto the stack on line 3. During the function epilogue in Figure 2(b), the signed return address is popped from the stack on line 1 and is verified (using the same salt it was signed with) on line 3. On line 4, the verified return address is used to return to the caller. If verified return address is valid (meaning the pointer was tampered with by attackers), this jump will be successful. Otherwise, an exception will occur.

According to the Apple Platform Security [33], in addition to protecting return addresses, macOS extensively uses Pointer Authentication to protect various other data structures, including C++ vtable pointers, vtable entries, and Objective-C method caches.

2.3 Micro-architectural Side Channels

Micro-architectural side channels are serious security threats to modern computing systems. It has been demonstrated that almost any shared micro-architectural structures, including caches [44, 51, 53, 69], TLBs [28], functional units [3], and network-on-chips [52], can be used to leak security-sensitive information. In this paper, our PoC attacks use TLB-based side channels [28], which share a lot of similarities with the extensively-studied cache-based side channels.

Both TLB and cache side channels exploit the timing differences of memory accesses. Flush+Reload [69], Evict+Reload [42], and Prime+Probe [44] are three commonly-used attack strategies. Among the three attack strategies, Prime+Probe has the least amount of restrictions, as both Evict+Reload and Flush+Reload require sharing memory between the attacker and the victim.

An example of the Prime+Probe attack targeting a TLB works as follows. The attacker starts by constructing a group of addresses that map to the same set of a TLB with at least as many lines as the associativity of the TLB. This group of address is usually referred to as an "*eviction set*". The attacker then repeats the following operations: 1) "primes" a TLB set by fully occupying the TLB set using addresses from the eviction set; 2) waits for the victim to either perform an access to the same set or do nothing; 3) "probes" the TLB set by re-accessing all the addresses in the eviction set and measuring their latency. If the measured latency is high, it means the victim has accessed the TLB set during step 2 (as the increased latency means that the attacker's eviction set was evicted from the TLB); otherwise, it means the victim has not performed the access.

2.4 Speculative Execution Attacks

Speculative execution or transient execution attacks are a type of information leakage attack that exploits the micro-architectural side effects of instructions which are speculatively executed but are squashed later. There exist numerous variants of speculative execution attacks [37, 43, 58, 63]. Meltdown [43] exploits the side channels caused by speculatively-executed instructions after an hardware exception. The class of Spectre attacks [15, 37, 38] exploit a broader attack surface and are considered much more difficult to mitigate comprehensively. Spectre v1 exploits misprediction of conditional branches [37], Spectre v2 exploits misprediction of indirect branch targets [37], and other variants exploit return address misprediction [38, 46] and speculative store-to-load forwarding [32].

3 THREAT MODEL

We consider an attacker whose ultimate goal is to conduct a control-flow hijacking attack on a PA-enabled victim using the PACMAN attack as a stepping stone. To this end, the goal of the PACMAN attack is to construct a *PAC oracle*, offering the capability to precisely distinguish between a correct PAC and an incorrect PAC for a given pointer without causing any crashes. Specifically, before the attacker modifies a protected pointer to a different address chosen by the attacker, the attacker first uses the PAC oracle to brute-force all possible PACs and find the matching PAC for the chosen address. The attacker then bypasses the protection of Pointer Authentication by modifying the protected pointer using the matching PAC and hijack the control flow of the victim.

Our threat model has the following assumptions about the victim and the attacker. First, there exists an exploitable memory corruption vulnerability in the victim program, which allows the attacker to write to some memory locations in the victim process. This assumption aligns with the threat model of PA, and matches the statement from the open-source XNU kernel used by macOS.[2] Second, the victim is protected using Pointer Authentication and there exists PACMAN gadgets in the victim execution context. We show that such gadgets are commonly present in PA-enabled codebases in Section 4.3. Third, the attacker is able to perform a micro-architectural side-channel attack on the machine running the victim. This requires the attacker to have access to a high-resolution timer and use the timer to measure latencies of micro-architectural events.

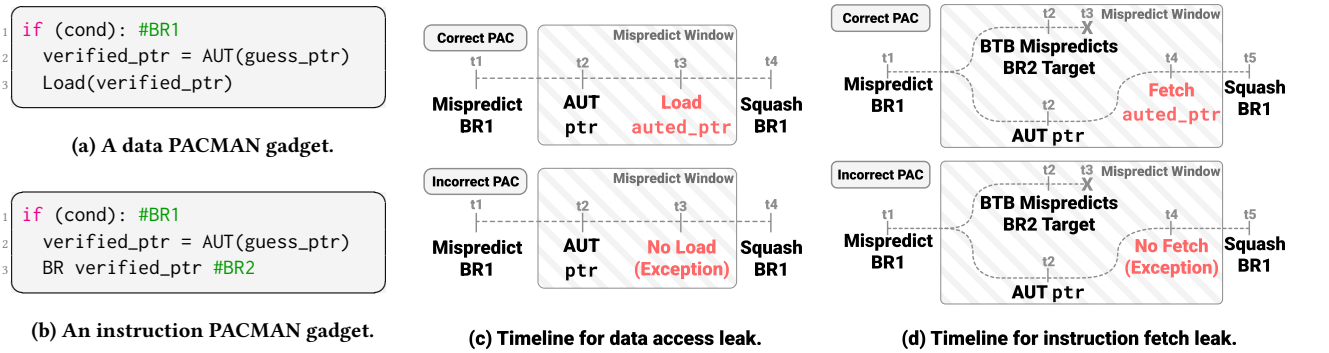
Throughout this paper, we consider an attack scenario across privilege levels, where the attacker is an unprivileged userspace application and the victim is the operating system kernel. The attacker can brute-force PACs for pointers signed using the kernelspace secret keys and perform control-flow hijacking attacks against the kernel.

4 THE PACMAN ATTACK

We propose the PACMAN attack, an attack that combines memory corruption attacks and speculative execution attacks. An attacker can use the PACMAN attack to construct a PAC oracle to distinguish between a correct PAC and an incorrect PAC for an arbitrary pointer. The key insight of the PACMAN attack is to use speculative execution attacks to leak PAC verification results stealthily via micro-architectural side channels without causing crashes. We find several code patterns to be effectively used for this purpose, and we call these code patterns *PACMAN gadgets*.

A PACMAN gadget consists of two operations, i.e., a *verification* operation and a *transmission* operation. The verification operation speculatively verifies the PAC of a given pointer and produces the verification result, and the transmission operation speculatively transmits the verification result via a micro-architectural side channel. Specifically, the verification operation is usually performed by a pointer authentication instruction (`AUT`), which outputs a valid pointer if the verification succeeds and an invalid pointer otherwise. The transmission operation can be performed by a memory load/store instruction or a branch instruction that takes the output pointer as its target address. If a correct PAC is used, the transmission operation accesses a valid address, resulting in observable micro-architectural side effects. Otherwise, the transmission operation causes a speculative exception due to accessing an invalid address. Note that, both of the verification and the transmission operations are performed under the shadow of a mis-speculated branch, and thus only cause micro-architectural side effects and will not trigger any architectural visible impacts (such as crashes).

[2]The XNU kernel [6] explicitly states that "Pointer authentication's threat model assumes that an attacker has found a gadget to read and write arbitrary memory belonging to a victim process, which may include the kernel."



(a) A data PACMAN gadget.

(b) An instruction PACMAN gadget.

(c) Timeline for data access leak.

(d) Timeline for instruction fetch leak.

Figure 3: PACMAN gadgets and their corresponding execution timelines.

Depending on the type of instructions used in the transmission step, we further divide PACMAN gadgets into two categories, i.e., data PACMAN gadgets and instruction PACMAN gadgets. In this section, we first explain the different behaviors of the two types of gadgets in Section 4.1 and Section 4.2. We then show that both types of gadgets are common in PA-enabled programs in Section 4.3. We put everything into the full picture by describing an end-to-end example of using the PACMAN attack to assist a control-flow hijacking attack in Section 4.4.

4.1 Data PACMAN Gadget

A data PACMAN gadget transmits PAC verification results via speculative data accesses. Figure 3(a) shows a minimal representation of a data PACMAN gadget, which consists of three instructions: a branch instruction, a pointer authentication instruction, and a memory access instruction. The pointer authentication instruction (line 2) takes an attacker-supplied signed pointer, denoted as `guessed_ptr`, as an operand, and verifies the correctness of the PAC using a secret key and the current context (not shown in the pseudocode for succinctness). The verification output, denoted as `verified_ptr`, is then used as the target address of a load instruction (line 3). Both the pointer authentication instruction and the load instruction are executed under the shadow of a mis-speculated branch.

We show the corresponding timeline of executing the data PACMAN gadget and highlight how the PAC verification result is leaked in Figure 3(c).

- At `t1`, the branch mis-speculation starts.
- At `t2`, the processor speculatively executes the pointer authentication instruction (`AUT`) on the guessed pointer (`guessed_ptr`). The instruction generates a valid address if the guessed PAC is correct and an invalid address otherwise.
- At `t3`, if the verified pointer (`verified_ptr`) is a valid address, the load instruction will be issued to the memory hierarchy, resulting in observable micro-architectural side effects. Otherwise, the load will not be issued to the memory hierarchy and instead will cause a speculative exception.
- At `t4`, the processor squashes the speculatively executed instructions, and suppresses any speculative exceptions. As a result, even if the guessed PAC is incorrect, the program will not crash.

Attack Variations. The gadget shown in Figure 3(a) is a minimal representation of the data PACMAN gadget. Other instructions between the verification and transmission instructions, such as arithmetic instructions, can exist without affecting the attack. The transmission operation can be either a load or store instruction, as long as the processor issues store requests speculatively.

Our attack relies on using micro-architectural side channels to distinguish whether a memory access or an exception occurred at `t3`. Memory operations can leave side effects on various micro-architectural structures, such as caches [30, 44, 51, 69], TLBs [28], DRAM row buffers [54], and Network-on-chips [52]. Our attack is general enough to work with a wide range of micro-architectural side channels.

4.2 Instruction PACMAN Gadget

An instruction PACMAN gadget transmits PAC verification results via speculative instruction fetches. Note that, compared to the data PACMAN gadget, the instruction gadget has an additional requirement on the targeted processor for it to work, that is, the targeted processor must use a speculation mechanism that can *eagerly squash* nested branches. This behavior is common to many aggressively optimized out-of-order processors, including the Apple M1 SoC.

Figure 3(b) shows a minimal representation of an instruction PACMAN gadget, containing three instructions, a conditional branch instruction (denoted as `BR1`), a pointer authentication instruction, and an indirect branch instruction (denoted as `BR2`). Similar to the data PACMAN gadget, the pointer authentication instruction verifies the correctness of the PAC of an attacker-supplied pointer and outputs a verified pointer. The indirect branch can be a call, jump, or return instruction that uses the verified pointer as the target address.

We show the corresponding timeline of executing the instruction PACMAN gadget and explain why eager squash is necessary for the attack to work in Figure 3(d).

- At `t1`, the branch mis-speculation starts.
- At `t2`, the pointer authentication instruction (`AUT`) is executed. However, before the verified pointer is generated, the processor makes a prediction on `BR2`'s target using

the branch target buffer (BTB). This is a common behavior on modern processors, as these processors use aggressive branch prediction and speculate across *nested* branches.

- At `t3`, the execution of the `AUT` instruction completes and the verified pointer is generated. The processor detects a branch misprediction on the inner branch `BR2` and eagerly squashes `BR2` and any other instructions coming after `BR2` in program order.
- At `t4`, after the eager squash is done, the processor attempts to fetch instructions from the actual target address of `BR2`. If the attacker guessed the PAC correctly, `verified_ptr` will be a valid pointer and the instruction fetch will happen. Otherwise, the fetch will not be issued to the memory hierarchy and a speculative exception will be generated.
- At `t5`, the outer branch (`BR1`) is squashed, suppressing speculative exceptions if there is any.

To summarize, there are two constraints when using the instruction PACMAN gadgets to leak PAC verification results. First, the instruction PACMAN gadget only works on processors that support eager squashing of nested branches. Following the example above, if at `t3`, the processor decides to not squash `BR2`, the verified pointer (`verified_ptr`) would never be used at all and thus the verification result would not be leaked.

Second, the attacker needs to be able to distinguish between the side effects caused by fetching the verified pointer and the BTB prediction. Specifically, since the BTB predicted addresses are always loaded, for the attack to work, accessing the verified pointer must cause extra micro-architectural side effects in addition to fetching the BTB predicted addresses. For example, if the cache-based Prime+Probe attack is used, the BTB prediction and the verified pointer should map to different cache sets. In our case, as we use TLB-based side channels in our proof-of-concept attack (Section 8), our attack works when the BTB prediction and the verified pointer are located in different pages.

4.3 Gadget Detection

We perform a preliminary analysis on how commonly PACMAN gadgets exist in the XNU kernel. We built a static analysis tool using Ghidra's scripting API [50]. The tool starts by finding conditional branches in the kernel, and then inspects 32 instructions in both branch directions searching for PACMAN gadgets. We consider a PACMAN gadget is found if the destination register of a verification instruction (`aut`) matches the source register of a transmit instruction (a memory access or a branch instruction).

We test our tool on the release version of the XNU kernel used by MacOS 12.2.1 (`xnu-8019.80.24`) and find 55,159 potential PACMAN gadgets, of which 13,867 are data PACMAN gadgets, and 41,292 are instruction PACMAN gadgets. There is an average distance of only 8.1 instructions between the conditional branch and the transmit instruction.

Our analysis is incomplete given that we only search for 32 instructions after conditional branches and we only track data-dependence via registers, not memory. We envision that more gadgets can be found with a comprehensive analysis. Thus, we conclude PACMAN gadgets are readily discoverable in large PA-enabled codebases.

```
1  struct obj_t {
2      char buf[10];
3      void (*fp)(void);
4  };
5
6  void vulnerable_syscall(str, cond){
7    obj = new obj_t;
8
9    memcpy(obj.buf, str, strlen(str)); //buffer overflow
10
11   if (cond) { // instruction PACMAN gadget
12     auted_fp = AUT(obj.fp);
13     call auted_fp;
14   }
15 }
```

Listing 1: An end-to-end attack example.

4.4 An End-to-End Illustrative Example

We provide an end-to-end example to show how to use the PACMAN attack to bypass Pointer Authentication and assist a control-flow hijacking attack. For illustration purpose, we use a simplified example below. We additionally demonstrate a proof-of-concept attack targeting real-world code patterns, e.g., the C++ method dispatch process, in Section 8.3.

Listing 1 shows the victim code. The struct `obj_t` contains a buffer and a function pointer `fp`, where `fp` is protected by Pointer Authentication. The victim code contains a buffer overflow vulnerability on line 9 and an instruction PACMAN gadget on line 11-14. We assume the attacker is able to 1) trigger the buffer overflow by controlling the length of the input string `str` to overwrite the function pointer `fp`, and 2) control the direction of the branch (line 11) by setting the boolean variable `cond`. The attacker's ultimate goal is to modify the function pointer `fp` with its PAC to a chosen address `fp_jop` pointing to a JOP gadget, and successfully divert the control flow of the victim to execute the JOP gadget without causing crashes. Since the function pointer is protected using Pointer Authentication, if we simply overwrite the pointer, the PAC verification operation (line 12) will fail.

The end-to-end attack consists of two steps. First, the attacker constructs a PAC oracle and bruteforces the correct PAC for the address of the JOP gadget. Following the example above, the attacker starts by training the branch (line 11) to be taken. At this stage, the attacker does not trigger the buffer overflow vulnerability and sets the variable `cond` to be true. The training process also trains the BTB entry for the indirect branch (line 13) to match the original value of `fp`. After the training process, the attacker speculatively executes the PACMAN gadget to find the correct PAC for the JOP gadget address. This requires the attacker trigger the buffer overflow and overwrite the function pointer `<PAC,fp>` with the address of the JOP gadget and a guessed PAC `<PAC_guessed,fp_jop>`, while setting the `cond` to be false. Using a micro-architectural side-channel attack to detect the side effects of the above execution, the attacker can determine whether the guessed PAC is correct or not. The attacker repeats this process until the matching PAC is found.

Second, the attacker triggers the control-flow hijacking attack. After the matching PAC is found, this step follows the standard

memory corruption attack. The attacker triggers the buffer overflow, overwriting the function pointer using the address of the JOP gadget and its matching PAC. Besides, the `cond` variable is set to be true so that the function call can execute and commit. Since `obj.fp` has now become `fp_jop` with a matching PAC, the PAC check (line 12) can be successfully passed, and the control flow will be diverted to the attacker's chosen gadget (line 13).

5 ATTACK PLATFORM

We demonstrate our attacks on the Apple M1 SoC (sometimes shortened to "M1"), which is the first `aarch64` desktop processor released to market that supports the ARM v8.3 architecture extensions. It is a non-trivial task to perform microarchitectural attacks on Apple platforms due to the closed-source and undocumented nature of both the macOS operating system and the M1 processor.

Apple M1 SoC. The Apple M1 is a desktop processor released in late 2020 [5]. The processor uses the `aarch64` architecture (sometimes referred to as `ARM64`) and supports the ARMv8.3 extensions, including PA [55]. The `aarch64` architecture separates execution context privilege level into four different exception levels. EL0 is the least privileged execution mode, and is where usermode programs execute. EL1 is the supervisor privilege level, and is where the kernel executes. [3] We provide proof-of-concept demonstrates of the PACMAN attack that can leak PAC verification results across privilege levels from the kernel to the userspace.

The M1 processor uses a big.LITTLE design [10], with 4 performance cores (also referred to as p-cores) and 4 efficiency cores (also referred to as e-cores). We target our attack on the performance cores, which have provided a more reliable attack surface due to a higher degree of speculation.

macOS and XNU Kernel. Apple Mac computers run macOS, which is a closed-source platform built on the open-source Darwin operating system and XNU kernel. The XNU kernel supports loadable kernel extensions, called *kexts*. Our reverse engineering effort utilizes kexts extensively for modifying the system to perform detailed analysis, as well as probing model-specific registers (MSRs) to read hardware configuration details.

Existing Reverse Engineering Efforts. The community has put forth significant effort towards understanding the undocumented internals of the M1 SoC. The Asahi Linux development team has been working to port Linux to the M1 SoC, and has revealed many of the undocumented hardware details specific to M1[2]. Additionally, several efforts have been made to reverse engineer the M1 micro-architecture with a focus on core pipeline details [24, 34, 39]. Unfortunately, none of these open-source reverse engineering efforts have uncovered sufficient details about the memory hierarchy of the M1 processor to enable our attack.

6 REVERSE ENGINEERING TOOLS

Micro-architectural side-channel attacks on Apple platforms are under-explored due to the closed-source and undocumented nature of both the software and the hardware. In this paper, we make significant contributions in expanding micro-architectural side-channel attacks to Apple processors by providing two sets of useful tools. First, we provide a comprehensive analysis of the timers on the M1 and construct two high-resolution timers accessible to EL0. Second, we develop PacmanOS, a bare-metal hypervisor that runs on M1 and enables noiseless reverse engineering experiments. We expect these tools to unblock the community from conducting research on existing and future Apple Silicon devices.

	MSR	EL0 Enabled?
System Counter (24 MHz)	`CNTPCT_EL0`	Yes
ARM Cycle Count Register	`PMCCNTR_EL0`	No*
Apple Performance Counter	`PMC0`	No
Multi-thread Counter	—	Yes

Table 1: Summary of timers on M1. (*This counter does not exist on M1.)

6.1 High-Resolution Timers

We summarize our investigation of the timers on M1 in Table 1.

First, there exists a system counter, `CNTPCT_EL0`, which is shared by all cores and is accessible from EL0 by default (tested on macOS Big Sur 11.5). However, this counter operates at a very low frequency, i.e., 24 MHz according to the Counter-timer Frequency register (`CNTFRQ_EL0`). This frequency is at the order of hundreds of CPU cycles and is too low to measure the precise latency differences required to properly reverse engineer the microarchitecture.

Second, most ARM Cortex processors are equipped with performance monitoring units (PMU) which include a Cycle Count Register (`PMCCNTR_EL0`). This cycle count register is extensively used in prior work on reverse engineering ARM processors [18, 42]. However, the M1 SoC does not use the standard ARM PMU, so this counter is unavailable on M1 at any privilege level.

With no high-resolution timers available in userspace, we construct the following two timers by 1) exploring Apple's undocumented and proprietary performance counter system and 2) building our customized timer counter using multi-thread execution.

Exposing Apple Performance Counters to Userspace. Apple has undocumented platform-specific performance counter registers, including `PMC0` and `PMC1`, where `PMC0` (`S3_2_c15_c0_0`) counts cycles and `PMC1` (`S3_2_c15_c1_0`) counts instructions.[4] However, these counters are only accessible in the kernel. To aid in the reverse engineering effort from userspace, we use a kext (XNU kernel extension) to configure the performance counter control register `PMCR0` (`S3_1_c15_c0_0`) and make `PMC0` readable from userspace. Since this approach requires loading a kext, we only use these counters to aid in the reverse engineering process, and do not use them in the actual attack.

Building A Custom Multi-thread Timer. We leverage multi-threaded execution and shared memory to build a timer that is accessible in userspace and does not require installing kernel extensions. This approach has been used in assisting attacks on ARM mobile devices [42] and on integrated CPU-GPU systems [20].

[3] Operating systems can also run in EL2 when Virtualization Host Extensions (VHE) is enabled.

[4] See `osfmk/arm64/monotonic_arm64.c` in the XNU kernel [8].

```
//a shared variable
volatile uint64_t counter;

void timerthread() {
    while(1) {
        counter++;
    }
}
```

(a) The dedicated timer thread.

```
isb
ldr time1, [counter_addr]
isb
// operations to time
isb
ldr time2, [counter_addr]
isb
sub latency, time2, time1
```

(b) The measuring thread.

Figure 4: The multi-thread timer.

Our customized timer requires two threads sharing a variable called `counter` (pseudocode in Figure 4). One thread works as a dedicated timer thread which increments the shared variable in an infinite loop. The other thread works as a measuring thread which reads the `counter` value before and after the operations to time. We use the `isb` instruction (Instruction Synchronization Barrier) to enforce the ordering of these operations.

We only use the `isb` instructions in the measuring thread, not in the dedicated timer thread for the following reason. Using the serialization instruction will slowdown the speed of incrementing the counter variable, and thus decrease the resolution of our timer. We tried both versions of the timer with and without `isb` instructions. We find that the absence of serialization barriers introduces slightly higher variance of the timing measurements, but significantly increases the resolution. Overall, we see higher reliability with the dedicated timer thread without the serialization barriers.

6.2 PacmanOS

As part of our reverse engineering effort, we designed PacmanOS, a bare-metal execution environment written entirely in Rust that can boot directly on M1. When PacmanOS boots, it sets up a minimal execution environment and runs a single experiment directly on the bare hardware. PacmanOS allows researchers to have complete control of the hardware, e.g., configuring and probing arbitrary model-specific registers (MSRs), creating arbitrary paging configurations, and performing noiseless reverse engineering experiments, without interference from other system software. Other bare metal hypervisor environments for M1 exist [2]; however, we wanted to design a minimum viable environment with complete hardware control for running noiseless experiments. We believe PacmanOS will serve as a useful tool for further study of M1 and future Apple Silicon devices.

7 REVERSE ENGINEERING

We reverse engineer the memory hierarchy of the M1 SoC with a focus on the TLBs. We start by presenting the basic memory hierarchy information obtained using a kernel extension (kext), followed by detailed reverse engineering findings of the TLB hierarchy on M1. To the best of our knowledge, we are the *first* to report these micro-architectural details.

	Level	Ways	Sets	Line Size	Total Size
p-core	L1I	6	512	64 B	192 KB
	L1D	8	256	64 B	128 KB
	L2	12	8192	128 B	12 MB
e-core	L1I	8	256	64 B	128 KB
	L1D	8	128	64 B	64 KB
	L2	16	2048	128 B	4 MB

Table 2: Cache configurations on M1 obtained via reading system registers.

7.1 Basic Memory Hierarchy Information

The M1's memory hierarchy has 2 levels of caches whose sizes are different for the p-cores and e-cores. Using a kernel extension, we read the CPU cache configuration registers to reveal the architecturally visible cache specification, shown in Table 2.

The private L1 instruction cache , L1 data cache, and shared L2 cache sizes on the p-core side are 192KB, 128KB, and 12MB respectively. The corresponding cache sizes on the e-core side are 128KB, 64KB, and 4MB respectively. There also exists a 16 MB system level memory-side cache shared across all components on the SoC. The cache line size of the L1 data and instruction caches is 64 bytes, and the line size of the L2 caches is 128 bytes.

The macOS 12.2.1 kernel uses 48-bit virtual addresses and 16-bit PACs with 16KB pages.

7.2 L1 Data TLB and L2 TLB

We focus on reverse engineering the memory hierarchy for the p-cores. Unlike Linux which provides flexible commands to pin a process to user-specified cores (via the `taskset` command), macOS does not offer such interface to an unprivileged user. Instead we use the `pthread_set_qos_class_self_np` API to suggest the kernel to schedule an experiment on a p-core. In this section, we report latency numbers using the Apple Performance Counter.

We start by analyzing the data access behaviors. The data access latency can be influenced by multiple factors, including the caches and TLBs. We first present our results on the TLBs, and then discuss the interaction between the TLBs and the caches.

Experiments. To reverse engineer the TLB parameters and exclude the impacts from the caches, we conduct the following experiment:

(1) Load an address x.
(2) Load N addresses that could potentially form an eviction set for a data TLB set without causing cache conflicts.
(3) Reload address x and measure the reload latency.

The address in step (2) is derived using the following formula:

$$\text{Addrs}[i] = x + i \times \text{stride} + i \times 128\text{B}, \text{ where } 1 \le i \le N$$

Recall that the M1 platform uses 64B or 128B as cache line size. The term $i \times 128$B in the formula maps the addresses to different cache sets, avoiding cache conflicts during the experiments.

Analyzing TLB conflicts. We perform a space search to understand TLB conflicts by varying the value stride by multiples of 16KB and varying the number of addresses N from 1 to 30. In Figure 5(a), we report the median reload latency of accessing the address x

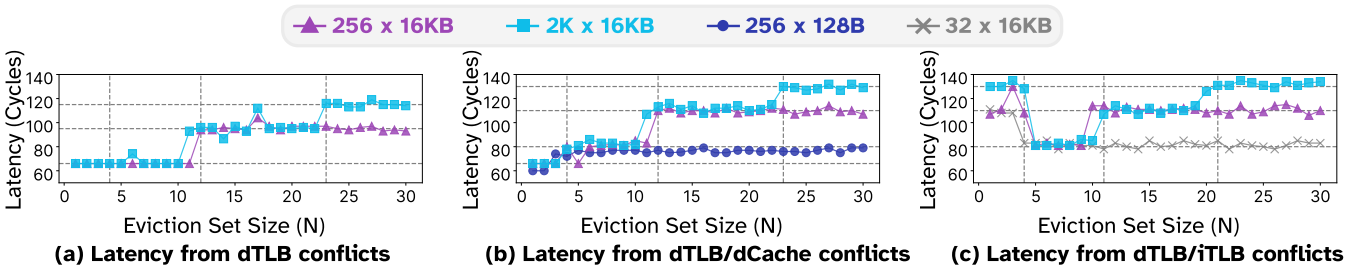


Figure 5: Access latency to the target address when varying the stride and the number of loads and branches executed. Labeled strides are the lowest strides where we observe the increase and decrease in access latencies.

across 1000 samples. For succinctness, instead of plotting a line for each stride value, we only show the lines of the lowest stride values that cause increases in the access latencies.

The access latency starts around 60 cycles, increases to around 95 cycles when stride $\geq 256\times$16KB and N $\geq$ 12, and further increases to 115 cycles when stride $\geq 2048\times$16KB and N $\geq$ 23. We hypothesize that the two latency jumps are caused by L1 dTLB conflicts and L2 TLB conflicts. It is very likely that the L1 dTLB is 12 ways with 256 sets and L2 TLB is 23 ways with 2048 sets if the TLBs are inclusive. Even though we are confident about the eviction parameters discovered by our analysis, we do not intend to conclude that the M1 chip follows our hypothesized design, because there may exist hidden structures in the micro-architecture that our analysis has missed. We conclude with the following findings:

> 1) To evict a page table entry from the L1 dTLB, we can create an eviction set with 12 or more addresses with a stride of $256\times$16KB.
> 2) To evict a page table entry from the L2 TLB, we can create an eviction set with 23 or more addresses with a stride of $2048\times$16KB.

Interactions between TLBs and caches. To understand how TLB conflicts and cache conflicts affect data access latency, we perform a similar experiment by changing the address calculation in step (2) to the formula below.

$$\text{Addrs}[i] = x + i \times \text{stride}, \text{ where } 1 \leq i \leq N$$

We vary the stride value to make it a multiple of 128B and we show the observed latencies in Figure 5(b).

The access latency starts around 60 cycles as before, increases to around 80 cycles when stride $\geq 256\times$128B and N $\geq$ 4, further increases to around 110 cycles when stride $\geq 256\times$16KB and N $\geq$ 12, and finally jumps to around 130 cycles when stride $\geq 2048\times$16KB and N $\geq$ 23. We hypothesize that the first jump (N = 4) is caused by L1 data cache [5] conflicts and the 80-cycle latency corresponds to L2 cache hits and L1 dTLB hits. We hypothesize that the next 2 jumps are caused by L1 dTLB conflicts and L2 TLB conflicts, because the 2 jumps happen when N = 12 and N = 23, which match what we have observed for TLB conflicts.

[5]Note that the observed L1 data cache associativity is half of the number reported by the system registers (Table 2). In this paper, we use the TLBs as the communication channel and defer the reverse engineering of the caches as future work.

7.3 L1 Instruction TLB

We next investigate the instruction access behaviors and reverse engineer the instruction TLB parameters. We additionally observe important interaction patterns between the instruction TLB and the data TLB which we find to be essential for conducting the PACMAN attack.

To create desired instruction access patterns, we allocate a large JIT memory region and fill the region with branch instructions where the target of each branch is properly configured to achieve the desired access pattern. Note that, an address in this memory region can be both fetched as an instruction address and accessed as a data address. Our experiment is then conducted as follows.

(1) Reset the L1 dTLB and L2 TLB by loading 23 data eviction addresses with a stride of $2048\times$16KB.
(2) Branch to a target address x so that the address will be fetched into the processor as an instruction.
(3) Execute N branch instructions that could potentially form an eviction set for an instruction TLB set. We call this group of addresses as our *instruction eviction set*.
(4) Load the target address x as data and measure the reload latency.

The address in step (3) is derived using the following formula:

$$\text{Addrs}[i] = x + i \times \text{stride} + i \times 128\text{B}, \text{ where } 1 \leq i \leq N$$

A subtle thing to note here is that the target address x is accessed as an instruction in step (2), but accessed as data in step (4). We design the experiment in this way because measuring data access latency is much more reliable than measuring instruction fetch latency and does not suffer from the noise caused by branch predictors and aggressive instruction prefetchers. We then vary the stride of the branches in step (3) by a multiple of 16KB and the number of branch instructions N from 1 to 30 to obtain Figure 5(c).

Analyzing iTLB Conflicts. When the number of the instruction eviction addresses is small (N $<$ 4), we observe a high access latency above 110 cycles. The access latency then drops from above 110 cycles to around 80 cycles when stride $\geq 32\times$16KB and N $\geq$ 4. The observation that the access latency decreases when we increase the number of eviction addresses contradicts the common understanding of how iTLBs and dTLBs are organized. Given that we consistently observe this phenomenon, we hypothesize that the L1 iTLB and L1 dTLB follow a design explained below.

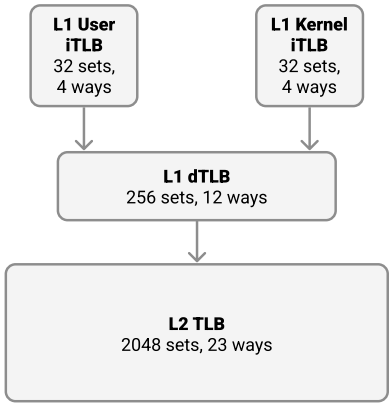


Figure 6: The TLB hierarchy on M1.

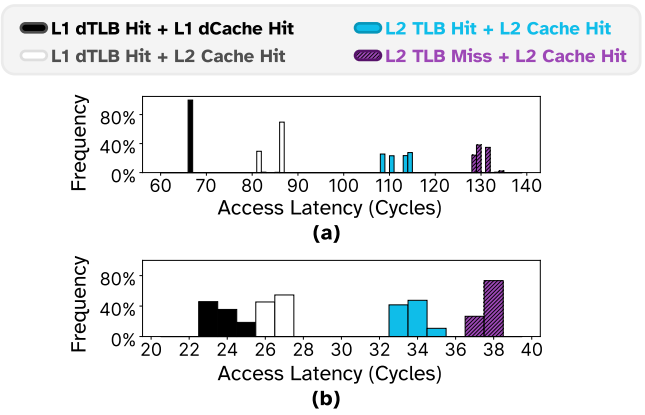


Figure 7: Latency measured using (a) Apple Performance Counter, and (b) our customized multi-thread timer.

First, the L1 iTLB and L1 dTLB are separate structures and the L1 iTLB content is not visible from the load and store ports. This would explain the high latency we observe when measuring the latency of loading the target address as data while the corresponding page entry is in the L1 iTLB. Second, the L1 dTLB serves as a non-inclusive backing-store of the L1 iTLB. Increasing the L1 iTLB eviction set causes the page table entry to be evicted from the L1 iTLB to the L1 dTLB. Thus, the entry becomes visible from the load and store ports on the data-fetch side, and we start to observe L1 dTLB hit latencies. Third, the L1 iTLB has 4 ways with 32 sets, as indicated by the line labeled with stride = 32×16KB in Figure 5(c).

Back to Figure 5(c), we observe the access latency increases from 80 cycles to around 110 cycles when stride ≥ 256×16KB and N ≥ 12, and further increases to around 130 cycles when stride ≥ 2048×16KB and N ≥ 23. These two increases completely match our observation in the previous experiments about L1 dTLB and L2 TLB (Section 7.2). We thus believe that once the a page table entry is evicted from the L1 iTLB to the L1 dTLB, it behaves the same as the other entries. Thus, we conclude with the following finding:

> 3) To evict a page table entry from the L1 iTLB, we can create an eviction set with 4 or more branch instructions with a stride of 32×16KB.

7.4 Summary of Reverse Engineering Results

We now summarize our reverse engineering results focusing on the information that is essential for conducting the proof-of-concept PACMAN attacks. First, as we will demonstrate the PACMAN attack across privilege levels, we repeat the reverse engineering experiments above to reverse engineer how TLBs are shared between userspace and kernelspace. Second, as we will use our customized timer in the PoC attacks, we evaluate the reliability of the customized timer and derive the threshold to distinguish between TLB hits and misses.

The TLB Hierarchy on M1. According to our reverse engineering results, each p-core on M1 has four separate TLB structures organized hierarchically as shown in Figure 6. The L1 iTLBs are *not shared* between the kernelspace and userspace, while the L1 dTLB and the L2 TLB are shared across privilege levels. Both the userspace L1 iTLB and the kernelspace L1 iTLB use the L1 dTLB as a non-inclusive backing-store. When a page table entry is evicted from one of the L1 iTLBs, it will be inserted into the L1 dTLB. We also include our reverse engineered parameters for each TLB structure in Figure 6. Note that we do not draw conclusions about the M1 design, since it is possible that Apple's design just happens to behave the same as our hypothesized design, but actually deviates from our description.

Latency and Threshold. We show the distributions of the measured memory access latencies via the Apple performance counter and our customized thread timer in Figure 7. Despite the differences between the two distributions, we see that both timing methodologies are able to clearly distinguish the memory access latencies as they hit or miss in different cache and TLB locations.

We can derive the threshold for distinguishing these accesses. For our customized timer (Figure 7(b)), an L1 dTLB hit is never beyond 27, while an L1 dTLB miss is never below 32. As such, the threshold to distinguish between an L1 dTLB hit and miss can be set to 30, to sufficiently monitor L1 dTLB accesses. This is also the threshold that we consistently used in the PoC attacks.

8 PROOF-OF-CONCEPT ATTACKS

We build our proof-of-concept (PoC) attacks step by step. First, we construct PAC oracles using the data PACMAN gadget and the instruction PACMAN gadget in Section 8.1. Second, we use the PAC oracle to brute-force PACs and further evaluate the speed and the accuracy of the brute-force attack in Section 8.2. Finally, we demonstrate a control-flow hijacking attack targeting the PA-enabled kernel in Section 8.3. All the PoC attacks are performed across privilege levels, where the attacker is an unprivileged userspace program and the victim is the operating system kernel.

8.1 PAC Oracles

We start by constructing PAC oracles across privilege levels. To set up the experiment, we install a kernel extension (kext) containing a

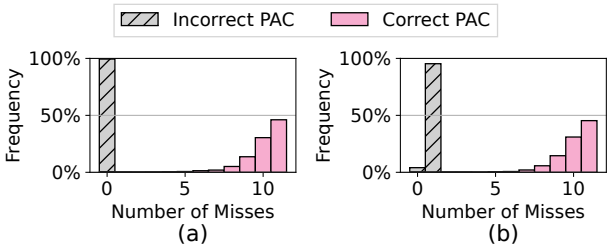


Figure 8: PAC oracle results. Number of observed L1 dTLB misses when leaking via (a) data PACMAN gadget, and (b) instruction PACMAN gadget.

PACMAN gadget that can be triggered via a syscall from userspace. To evaluate the accuracy of the PAC oracle, we pass a pointer embedded with a chosen PAC to the kext and monitor the TLB modulations caused by the syscall. In each run, we randomly decide whether to use the correct PAC or a randomly generated incorrect PAC. We use our customized multi-thread timer to report the side-channel attack results for 20,000 trials, where half of the trials used an incorrect PAC, and half used the correct PAC.

Leaking via Data Accesses. As the userspace and kernelspace share the L1 dTLB, we utilize Prime+Probe to monitor the L1 dTLB activities of the data PACMAN gadget. Specifically, we prepare an eviction set with 12 address with a stride of 256×16KB between each address. Besides, as we find that the Prime+Probe attack is sensitive to the TLB replacement policy, we additionally introduce a reset operation which can help significantly increase the attack accuracy. The attack process works as follows.

(1) Train the branch predictor in the data PACMAN gadget in the kernel to be taken for 64 times.
(2) Reset the TLB hierarchy by accessing 23 addresses that map to the same L2 TLB set. These addresses should not be part of the eviction set used for Prime+Probe.
(3) Prime the L1 dTLB set by accessing the eviction set.
(4) Trigger the PACMAN gadget by passing in the pointer with the PAC to test.
(5) Probe the L1 dTLB set by re-accessing the eviction set and report the number of L1 dTLB misses.

We show the attack results in Figure 8(a). When an incorrect PAC is used, in almost all cases (99.2%), we observe no L1 dTLB misses. When a correct PAC is used, we observe at least 5 misses for 99.6% of the time, which sufficiently indicates that our attack can reliably distinguish between correct and incorrect PACs.

Leaking via Instruction Fetches. There exists a key challenge to monitor the TLB modulations by the instruction PACMAN gadget. The challenge lies in the fact that as the the L1 iTLBs are not shared between the kernelspace and userspace (Section 7.4), the attacker in the userspace is unable to observe the instruction fetches performed by the victim in the kernelspace. We address this challenge by creating self-conflicts in kernelspace to evict a target TLB entry from the private L1 iTLB to the shared L1 dTLB, whose states can be directly monitored from the userspace.

Our cross-privilege attack via the instruction PACMAN gadget follows similar steps as leaking via the data PACMAN gadget and only differs in the last step. Specifically, we replace the step (5) with the two steps below.

(5) Evict the target TLB entry from the kernelspace L1 iTLB by fetching 4 instructions with a stride of 32×16KB between each of them. We make 4 system calls to perform these instruction fetches in the kernelspace.
(6) Probe the L1 dTLB set by re-accessing the eviction set and report the number of L1 dTLB misses.

We show the attack results using the instruction PACMAN gadget in Figure 8(b). Similar to the previous attack, when an incorrect PAC is used, we observe at most 1 accesses for 99.2% of the time, and when the correct PAC is used, we observe at least 5 misses for 99.8% of the time. Given that the two distributions are clearly distinguishable, we consider the PAC oracle highly reliable.

8.2 Brute-Force Attack

Attack Speed. We perform a preliminary speed test of using the PACMAN oracle to brute-force the correct PAC. When we train the conditional branch in the instruction PACMAN gadget for 64 times, it takes 2.69 milliseconds on average to test one PAC value. Considering that the M1 uses a 16-bit PAC, we estimate that it will take around 2.94 minutes on average to try all possible PAC values.

We find the brute-force attack speed is dominated by the syscall overhead during the training iterations. Therefore, the attack speed can be further increased if one can reduce the number of training iterations or reduce the kernelspace and userspace context switch overhead. We do note that this speed analysis result is preliminary, mainly because our tested syscall function is very short. The speed of the PACMAN attack in a real-world setup depends on how long the targeted syscall function takes.

Attack Accuracy. We evaluate the accuracy of the brute-force attack by testing every possible PAC value starting from `0x0` to `0xFFFF`. For each guessed PAC, we collect 5 samples and determine whether the corresponding PAC is correct based on the median miss count. For each brute-force experiment, we have three potential outputs: 1) True Positive when the correct PAC is found; 2) False Positive when an incorrect PAC is found; 3) False Negative when no PAC is found. Note that, we cannot tolerate false positives, since using an incorrect PAC can crash the system. However, our attack can easily tolerate false negatives, because when no PAC is found, the attacker can simply repeat the brute-force process until the correct PAC is found.

We repeat the brute-force experiments 50 times, under a noisy environment where we browse websites and make video calls. We observe consistent results for attacks using the data PACMAN gadget and the instruction PACMAN gadget. Specifically, we found the correct PAC in 90% of the experiments (45 out 50), while in the remaining 10% of the experiments (5 out of 50), no PAC value was found. Importantly, we observe no false positives. Since false negatives are tolerable, our evaluation result indicates perfect reliability.

```
1 //call an object's ith method
2 vtable_ptr = AUT(*object_addr); // obtain vtable pointer
3 fp = AUT(vtable_ptr[i]);   // obtain function pointer
4 call fp;                   // make a function call
```

Listing 2: C++ method dispatch process.

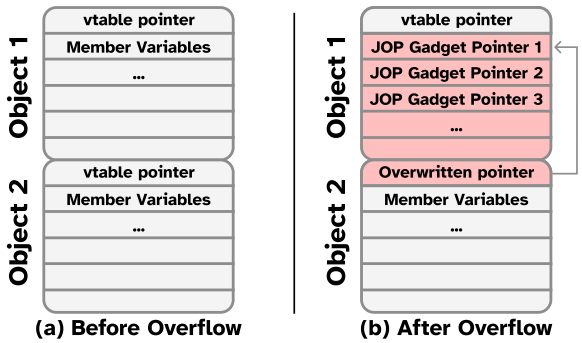


Figure 9: A jump2win attack.

8.3 Jump2Win Attack

We extend our PACMAN attack to assist a control-flow hijacking attack against the kernel via a single buffer overflow vulnerability. Specifically, we demonstrate a Jump2Win attack where we modify a function pointer protected using Pointer Authentication to the address of a `win` function which exists in the kernel.

We exploit the C++ method dispatch process [33] shown in Listing 2. In C++, the first 8 bytes of an object contain the vtable pointer which points to a list of function pointers for the object's methods. In a PA-protected program, the method dispatch process first loads the vtable pointer and verifies it using the data key (line 2). Next, the vtable is indexed to obtain the desired function pointer, which is verified using the instruction key (line 3) before the actual function call (line 4). In both verification operations, the salt is the object address plus a compile-time constant value.

Our attack works by exploiting a buffer overflow vulnerability to overwrite the vtable pointer in a victim object. We show the memory content before and after the buffer overflow in Figure 9. There are two objects allocated contiguously in memory, where the first member variable in `object1` is a buffer. The attacker overflows the buffer and overwrites the other member variables in `object1` as well as the vtable pointer in `object2` (the corrupted memory regions are highlighted). The attacker fills the buffer with a signed address pointing to the `win` function, and overwrites `object2`'s vtable pointer with a signed address pointing to the buffer (shown in Figure 9(b)). The PACs for the two signed addresses can be obtained using the PACMAN attack. Once the buffer overflow is done, when a function call in `object2` is triggered, the victim will load the buffer address as the vtable pointer, and use the address of the `win` function as the function pointer.

To show this attack pattern is viable, we created a kext with this buffer overflow vulnerability present, and implemented an attack against it from userspace. We successfully demonstrated that we can perform the attack above and trick the kernel to execute a `win` function.

9 COUNTERMEASURES

We discuss three potential directions in protecting against the PACMAN attack. The first direction is to explore *PAC-agnostic execution*. That is, modifying the microarchitecture or software to ensure that the verification results of PAC are never used under speculation. One approach is to pause speculative execution of using verified pointers for memory access instructions and branch instructions. This approach can be achieved by adding `fence/isb` instructions after any pointer authentication instructions. However, this approach can incur significant performance penalty as these instructions are very common programs protected using Pointer Authentication. Another approach is to always speculate a verified pointer assuming the verification to be successful. However, such an approach could introduce a Meltdown-style vulnerability by allowing speculatively de-referencing invalid pointers. Therefore, with such an approach, the system would have to incorporate some form of Meltdown mitigations [17] to fully patch the problem.

The second direction is to explore adapting prior work in defending against Spectre to defend against PACMAN. Invisible speculation mechanisms, such as InvisiSpec [68], SafeSpec [35], and Delay-on-Miss [57], should serve the purpose as they hide the side effects of any speculative load instructions. However, the recent speculative interference attacks [14] have demonstrated that these defense mechanisms can still indirectly leak the timing of load instructions. They also need to extend their protection from caches to TLBs considering our PoC attacks. Another set of mitigation mechanisms leverage information flow tracking [60, 62] to block leaking speculatively accessed secret, such as STT [70], NDA [67], and Dolma [45]. However, we find these work consider that the source of a taint starts from a load instruction. In our PACMAN attack, the taint starts from a pointer authentication instruction. A simple fix will be to re-purpose these mechanisms by marking the output of any pointer authentication instructions as tainted.

Lastly, as the PACMAN attack needs to work with a memory corruption bug to achieve exploitation, patching any memory corruption vulnerabilities would help.

10 RELATED WORK

We discuss related work of our PACMAN attack in the order of how close they are to our work. First, to the best of our knowledge, there exists only one work that also targets the synergies between memory corruption attacks and speculative execution attacks. Speculative Probe [27] demonstrates an attack that starts from a single memory corruption bug to probe every address in the kernel space and break kASLR [21]. We share the insights of using speculative execution for crash suppression. Similar to other speculative execution attacks, speculative probe focuses on leaking loaded content, while our attack leaks PAC verification results.

Second, there exist several attempts to break Pointer Authentication, documented in blog posts of Google Project Zero. For example, Brandon Azad [13] found a bug in the PA implementation of the Apple A12 processor: When giving an invalid pointer to a pointer signing instruction (the set of instructions starting with

pac), the instruction would generate a valid PAC, flip the top bit and return it. This bug has been patched in iOS and macOS system software [13]. However, it is not as trivial a task to defend against our PACMAN attack, as our attack exploits hardware speculative execution mechanisms which cannot be deactivated from system software.

Third, there have been a plethora of research proposals in the system and architecture community that extend the ARM PA feature increase its protection coverage or re-purpose it for detecting other vulnerabilities. For example, PAC It Up [41] incorporates PA into the LLVM compiler to monitor data accesses for misuse with minimal performance overhead, PTAuth [23] uses PA to detect temporal memory corruptions, Nasahl et. al [49] proposes to use PA to enforce CFI (control-flow integrity), and AOS [36] re-purposes PA to support memory safety protection of heap data. The security properties of those designs have been rigorously examined under the memory safety threat model. However, our work highlights the essential to revisit the security properties of these designs under a broader threat model, by taking speculative execution attacks into consideration.

11 CONCLUSION

We have presented PACMAN, a novel speculative execution attack against ARM Pointer Authentication. We have reverse engineered the TLB organizations on Apple M1 and have demonstrated multiple proof-of-concept attacks that work across privilege levels. We believe that this attack has important implications for designers looking to implement future processors featuring Pointer Authentication, and has broad implications for the security of future control-flow integrity primitives.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was funded in part by the NSF under grant CNS-2046359, and by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-20-1-0402.

REFERENCES

[1] 2001. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.
[2] 2021. Linux on Apple Silicon. https://asahilinux.org/about/.
[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[4] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (November 1996). http://www.phrack.com/issues.html?issue=49&id=14
[5] Apple Inc. 2020. Apple unleashes M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/
[6] Apple Inc. 2020. ARMv8.3 Pointer Authentication in xnu. https://github.com/apple/darwin-xnu/blob/xnu-7195.50.7.100.1/doc/pac.md
[7] Apple Inc. 2021. Introducing M1 Pro and M1 Max: the most powerful chips Apple has ever built. https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/
[8] Apple Inc. 2021. The Darwin Kernel 7195.81.3. https://github.com/apple-oss-distributions/xnu/tree/xnu-7195.81.3
[9] ARM Limited. 2021. ARM Architecture Reference Manual. https://developer.arm.com/documentation/ddi0487/ga.
[10] ARM Limited. 2021. ARM big little. https://www.arm.com/why-arm/technologies/big-little
[11] ARM Limited. 2021. Jump-oriented programming. https://developer.arm.com/documentation/102433/0100/Jump-oriented-programming.
[12] ARM Limited. 2021. Secure and Scalable Performance for Next-Generation On-The-Go Devices. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a78c.
[13] Brandon Azad. 2019. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html
[14] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*.
[16] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*.
[17] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*.
[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*.
[19] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. 98 (01 1998).
[20] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.
[21] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/.
[22] Úlfar Erlingsson, Yves Younan, and Frank Piessens. 2010. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer.
[23] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*.
[24] Andrei Frumusanu. 2020. The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test. https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested.
[25] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *JCEN* 8, 1 (2018).
[26] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goktas
[27] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.
[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.
[29] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham.
[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.
[31] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2014.43
[32] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.
[33] Apple Inc. May 2021. Apple Platform Security Guide. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf.
[34] Dougall Johnson. 2021. Apple M1 Microarchitecture Research. https://dougallj.github.io/applecpu/firestorm.html. Accessed on 24.11.2021.
[35] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*.
[36] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[38] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[39] Daniel Lemire. 2021. Memory access on the Apple M1 processor. https://lemire.me/blog/2021/01/06/memory-access-on-the-apple-m1-processor/.

[40] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium (USENIX Security 21)*.

[41] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*.

[42] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*.

[43] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[45] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*.

[46] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[47] Hector Marco-Gisbert and Ismael Ripoll. 2019. Address Space Layout Randomization Next Generation. *Applied Sciences* 9 (07 2019). https://doi.org/10.3390/app9142928

[48] MITRE. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[49] Pascal Nasahl, Robert Schilling, and Stefan Mangard. 2021. Protecting Indirect Branches against Fault Attacks using ARM Pointer Authentication. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[50] National Security Agency. 2022. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[52] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security*.

[53] Colin Percival. 2005. Cache Missing For Fun And Profit. In *BSDCan*.

[54] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*.

[55] Inc. Qualcomm Technologies. 2017. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[56] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012).

[57] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2020. Understanding Selective Delay as a Method for Efficient Secure Speculative Execution. *IEEE Trans. Comput.* 69, 11 (2020). https://doi.org/10.1109/TC.2020.3014456

[58] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *ISCA*.

[59] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*.

[60] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices* 39, 11 (2004).

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE.

[62] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*.

[63] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[64] Arjan van de Ven. 2004. Exec shield. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[65] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. 2012. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*. Springer.

[66] Yuan Wei, Senlin Luo, Jianwei Zhuge, Jing Gao, Ennan Zheng, Bo Li, and Limin Pan. 2019. ARG: Automatic ROP Chains Generation. *IEEE Access* 7 (2019). https://doi.org/10.1109/ACCESS.2019.2937585

[67] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[68] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2018.00042

[69] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[70] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.