Demystifying Pointer Authentication on Apple M1

Zechao Cai, Jiaxun Zhu, Wenbo Shen, Yutian Yang, and Rui Chang, *Zhejiang University and ZJU-Hangzhou Global Scientific and Technological Innovation Center;* Yu Wang, *Hangzhou Cyberserval Co., Ltd.;* Jinku Li, *Xidian University;* Kui Ren, *Zhejiang University and ZJU-Hangzhou Global Scientific and Technological Innovation Center*

https://www.usenix.org/conference/usenixsecurity23/presentation/cai-zechao

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Demystifying Pointer Authentication on Apple M1

Zechao Cai[1,2], Jiaxun Zhu[1,2], Wenbo Shen[1,2,✉], Yutian Yang[1,2], Rui Chang[1,2], Yu Wang[3], Jinku Li[4], and Kui Ren[1,2]

[1]*Zhejiang University, Hangzhou, China*
[2]*ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou, China*
[3]*Hangzhou Cyberserval Co., Ltd., Hangzhou, China*
[4]*Xidian University, Xi'an, China*

Abstract

Pointer Authentication (PA) was introduced by ARMv8.3 to safeguard the integrity of pointers. While the ARM specification allows vendors to implement and customize PA, Apple has tailored it on their hardware to protect iPhones and Macs with M-series chips. Since its debut, Apple PA has been considered effective in defeating pointer corruption. However, its details have not been publicly disclosed.

To shed light on Apple PA customization, this paper conducts an in-depth reverse engineering study focused on Apple PA's hardware implementation and usage on the M1 chip. We develop a reverse engineering framework and propose novel techniques to uncover and confirm our new findings.

Our study uncovers that Apple PA has implemented several hardware-based diversifiers to counter pointer forgery attacks across various domains, which is previously unknown to researchers outside of Apple. We further discover that the XNU kernel (the kernel used by iOS and macOS) incorporates nine types of modifiers for signing and authenticating pointers and customized key management based on Apple PA hardware. Based on our in-depth understanding of Apple PA, we perform a security analysis of PA-based control-flow integrity and data-flow integrity in the XNU kernel, identifying four attack surfaces. Apple has fixed these issues in a security update and assigned us a new CVE.

1 Introduction

Pointer Authentication (PA) is a security feature introduced by ARMv8.3 in 2016 to protect the integrity of pointers. It signs pointers with secret keys and authenticates the signature before dereferencing to detect pointer corruptions. ARM specification provides instructions for signature generation/authentication. It also specifies dedicated key registers [9], ensuring that secret keys are stored in the registers to prevent key leaks via memory. With PA protection, attackers can no longer forge legal pointers without knowledge of the secret keys.

✉Corresponding author.

However, ARM PA specification provides only five key registers, supporting five different key types. The same key register is used by signature generation/authentication instructions across different domains, such as different exception levels (ELs). This design flaw allows an attacker to forge a pointer signature in a domain (e.g., EL0 or user mode) and inject it into another domain (e.g., EL1 or kernel mode) to bypass authentication, leading to *cross-domain attacks*. As a result, how to implement and use PA securely still faces significant challenges. Apple is the first one implementing PA hardware and using it in commercial devices. Specifically, Apple introduced PA on the A12 chip in 2018. Since then, Apple increasingly relies on PA to protect iOS and macOS. Notably, all iPhones shipping after 2018 and all Macs with M-series chips are protected by PA [4].

The popularity of Apple PA-enabled devices has attracted the attention of security researchers from various teams, including Google Project Zero, Pangu, and Keen team. These researchers have analyzed the Apple PA usage and reported several security problems [16, 20, 22, 23, 41, 44]. However, these studies only analyzed partial PA usage, which is not comprehensive. Moreover, to the best of our knowledge, the PA hardware implementation has never been studied so far.

To address this research gap, we conduct an in-depth reverse engineering study to understand the PA hardware implementation and usage on Apple M1. We reveal the underlying hardware logic of Apple's PA system registers, key management, and defenses against cross-domain attacks. Additionally, we systematically analyze all PA instructions and examine the utilization of PA keys in the XNU kernel, which serves as the kernel for iOS and macOS [5].

To study Apple PA, we first need to know how PA is controlled. In the ARM specification, PA is controlled through system registers. Existing studies [14] show Apple added plenty of system registers on top of ARM specification. However, Apple has not publicly disclosed these registers, making it unclear if they are used for PA and what their functionalities are. Second, obtaining the actual values of PA keys is crucial for understanding Apple's PA mitigations against

cross-domain attacks. However, Apple's hardware-based protection prevents researchers, including Google Project Zero members, from accessing the actual PA key values. Consequently, acquiring the real PA key values is difficult. Third, analyzing the PA usage in XNU kernel requires debugging the system boot and changing kernel status at runtime. However, the official kernel debugger LLDB [33] is unable to debug the boot stage and change the status of the active kernel [19, 36].

To address these challenges, we propose hypervisor-based techniques for identifying PA system registers, reading/writing actual PA key values, and debugging the XNU kernel dynamically. We have developed a hypervisor-based PA reverse engineering framework specifically designed for Apple silicon by customizing a hypervisor named m1n1 [12]. The framework allows running macOS and Linux on top of m1n1 on an Apple M1 device, enabling PA hardware probing and software tracing to support the reverse engineering process.

Our findings. Using this reverse engineering framework, we reveal that Apple M1 introduced per-VM, per-key-type, per-boot diversifiers, and extra keys to defend against cross-VM, cross-key, cross-boot, and cross-EL attacks (generalized as cross-domain attacks). We also find that Apple introduced nine types of modifiers for signing and authenticating pointers and customized key management based on Apple PA hardware. Furthermore, we have conducted a security analysis of PA-based control-flow integrity (CFI) and data-flow integrity (DFI) implementation in XNU kernel. Through this analysis, we have identified four attack surfaces and validated 88 potential misuse cases. We have responsibly disclosed these attack surfaces to Apple. Apple has fixed these issues in a security update, assigned us CVE-2023-32424, and publicly acknowledged us on the security advisory.

In sum, this paper makes the following contributions.

- We develop a hypervisor-based PA reverse engineering framework and propose multiple new techniques.
- We reveal how Apple customizes PA hardware to mitigate cross-domain attacks.
- We reveal how PA is used in XNU kernel. We conduct a security analysis of PA usage to identify misuse cases.
- We plan to open-source our framework to facilitate further research on Apple silicon and improve PA security.

2 Background

2.1 Pointer Authentication

2.1.1 ARM PA Specification

ARM PA defines control registers, PA keys, and PA instructions for pointer signing and authentication. The control register `SCTLR_EL1` includes four bits that enable or disable PA [8]. ARM PA provides five types of PA key registers: `APIA`, `APDA`, `APIB`, `APDB`, and `APGA`. Each key register consists of two 64-bit system registers, representing the lower and higher 64 bits of the key. Access to key registers is restricted to privileged instructions such as `msr` (write) and `mrs` (read) instructions. For signature (a.k.a., *pointer authentication code or PAC*) computation, ARM recommends using the QARMA algorithm [15]. Additionally, ARM specifies `pac`/`aut` instructions for pointer signing and authentication.

Signing instructions take three inputs for computing PAC: the key, a pointer, and a modifier. ARM PA provides different signing instructions to specify PA keys and modifiers. For example, `pacia` instruction generates a PAC using `APIA` key and a user-define modifier, while `pacizb` instruction uses `APIB` key and zero as the modifier.

Authentication instructions authenticate signed pointers and convert them to canonical pointers upon successful authentication. For instance, `autia` instruction will authenticate a pointer signed by `pacia` instruction. If the authentication fails, the `aut` instruction will insert an error code to the upper bits or trigger an exception when FPAC [10] is implemented.

2.1.2 PA-based XNU Kernel Protection

XNU kernel uses PA to protect pointers and sensitive data. Besides using inline `pac/aut` instructions directly, XNU has also utilized wrapper functions of these instructions for signing and authentication. We term these inline PA assembly and these wrapper functions as *sign/auth interfaces*. These sign/auth interfaces are instrumented manually in XNU kernel code [2] or automatically by the compiler. However, there is no systematic study on how Apple uses the sign/auth interfaces to protect XNU kernel and analyze its security. Furthermore, since PA only supports five types of key registers, effective key management is crucial in PA to mitigate cross-domain attacks. However, the key management techniques employed by the XNU kernel remain undisclosed.

2.1.3 Existing Apple PA Analysis

Regarding PA hardware, researchers from Project Zero team [44] have discovered that Apple customizes the PA on iPhone XS by observing the outcomes of `pac` instructions. For convenience, we refer to the customized PA as "Apple PA" when discussing it further. Subsequent studies [14] have revealed that Apple PA is also present in the M1 chip and can be enabled through an Apple-specific system register. However, as far as our knowledge extends, no study has systematically disclosed the complete ISA definition of Apple's customization on PA or its security characteristics.

In terms of the PA-based XNU kernel protection, extensive analysis has been conducted by researchers [16, 22, 41, 44]. However, due to the lack of systematic examination of the PA hardware implementation on Apple Silicon, researchers have not comprehensively analyzed the effectiveness of PA-based protection mechanisms in the XNU kernel.

2.2 Virtualization Host Extension

The ARMv8.1 specification introduces Virtualization Host Extension (VHE) [6, 26] to allow OSes to run on EL2 (exception-level 2 or hypervisor mode). One important challenge for VHE is to redirect system register access of OSes

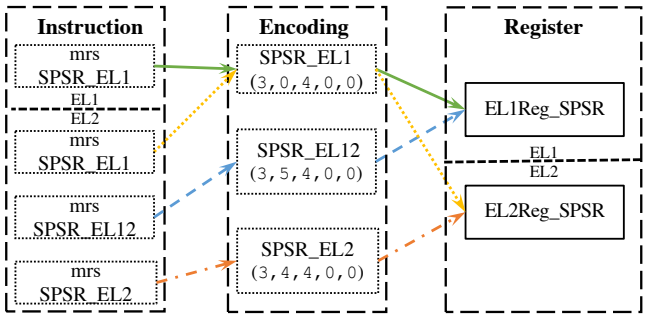


Figure 1: An example of *system register redirection* supported by ARM (Different line colors denote relationship between encoding and register for different instructions).

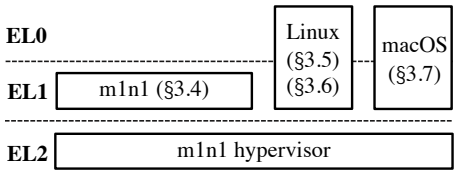


Figure 2: m1n1-based environments.

from EL1 to EL2 without software changes. VHE implements *system register redirection* to address the challenge by remapping the access instruction encoding to the actual registers. Specifically, an instruction uses *access instruction encoding* to specify a system register when accessing it. The encoding is in the form of `(op0,op1,CRn,CRm,op2)`. For example, in Figure 1, the register mnemonic `SPSR_EL1` specifies the encoding `(op0:3,op1:0,CRn:4,CRm:0,op2:0)`. For the same encoding `SPSR_EL1`, VHE redirects access from the host kernel (EL2) to the `SPSR` register on the host machine to achieve system register redirection. With VHE, different encodings can also map to the same register. Besides, VHE introduces an extra set of encodings with the suffix `_EL12` for the host kernel to access system registers in virtual machines.

We use the term *alias registers* to represent the encodings that map to the same register. Besides, we generally use the term *register* to refer to *encoding* for easy understanding. We distinguish *register* and *encoding* when necessary.

2.3 m1n1

m1n1 [12] is an open-source lightweight hypervisor for Apple silicon developed by AsahiLinux [11]. It offers access to privileged hardware features of the M1 platform, including privileged registers and instructions. This makes m1n1 suitable for testing and reverse-engineering hardware functionalities. Additionally, m1n1 supports running macOS virtual machines, enabling the observation and interaction with the macOS kernel. While m1n1 provides a range of capabilities, it is not designed for PA analysis. We customized it by adding exception handling, PA key reading/writing functionality, and other experimental code. These customizations enhance m1n1's functionality and enable researchers to analyze the PA mechanisms on the Apple M1 platform.

3 m1n1-based PA Reverse Engineering Framework

3.1 Goals

Apple has deployed a customized version of PA on their M1 processor, which differs significantly from the ARM PA. Our objective is to analyze Apple's hardware-level customization and conduct a comprehensive analysis of the PA-based protection in the XNU kernel. To accomplish this, we need to first identify the PA-related registers, and then dissect the behavior of PA instructions, and finally analyze how all PA instructions are utilized in the XNU kernel. To achieve these goals, we have identified four required capabilities.

3.2 Required Capabilities

RC 1. Identifying undisclosed Apple-specific system registers for PA. Previous works [11] show that Apple added many new system registers on top of ARM specification. We term these registers as *AppleReg*. Compared with ARM PA specification, Apple M1 introduces many PA-related *AppleReg*s. These *AppleReg*s are critical for understanding how PA works on M1. However, there is no official documentation about *AppleReg*s. Besides, most of the *AppleReg*-related code remains closed-source. It is a necessary capability to identify all PA-related *AppleReg*.

RC 2. Reading/writing actual PA key values. Apple introduces hardware customization to protect the actual key values. When Apple PA is enabled, the key registers cannot be read by `mrs x1, key_el1` instruction. Moreover, the key value set by `msr key_el1, x1` is not the actual value used in PAC computation. Researchers [44] found that PA key value is different by observing the result of `pac` instruction. However, they cannot obtain the actual key value due to Apple's hardware protection, preventing them from conducting a deeper analysis. It is necessary to bypass the hardware protection and be able to read and write the actual PA key values.

RC 3. Revealing the undisclosed behaviors of PA instructions. The behaviors of PA instructions are very different between Apple PA and ARM PA. As a result, we deduce that the hardware implementation of Apple PA is significantly different from ARM PA. Consequently, we need the capability to systematically and comprehensively dissect the behavior of PA instructions on M1.

RC4. Debugging system boot and modifying system state dynamically. Currently, most `pac` instructions are executed during boot-up. Meanwhile, we found that the operators of these `pac` instructions are propagated through the complex data flow, making it impossible to extract these operators using static analysis to achieve PA software analysis. To extract the operators of these `pac` instructions, we need to be able to debug the system at the booting stage (which LLDB cannot [19, 36]). Moreover, for attack surface validation, we need to suspend the system execution and change system states to observe subsequent behaviors.

3.3 Framework Overview

We design and implement a m1n1-based reverse engineering framework on an Apple M1 device to achieve the above-required capabilities. More specifically, we customized the system of an M1 device by running the m1n1 in EL2, as shown in Figure 2. On top of m1n1 in EL2, we further run three EL1 runtime environments. First, we run m1n1 in EL1 **(m1n1 + m1n1)** so that we can customize exception handling in both EL1 and EL2, allowing us to access arbitrary system registers without crashing the system. Second, we run Linux in EL1 **(m1n1 + Linux)** to provide a full-fledged running OS on M1. More importantly, we can modify the Linux kernel to develop our experiments conveniently. Third, we also run an unmodified macOS in EL1 **(m1n1 + macOS)** to trace and debug PA customization and usage. Based on the above three runtime environments, we propose four techniques (§3.4-§3.7) to achieve the RC1-4.

3.4 PA-related Apple-specific System Register Identification

In the XNU kernel binary, *AppleReg* is initialized with registers specified by ARM during kernel boots. Our key observation is that the XNU kernel test functions include numerous assertions for *AppleReg*. Besides, the system registers used to control the same hardware feature are often utilized within the same basic block or function. With these insights, we propose a technique combining static and dynamic analysis to achieve RC1. Specifically, our technique consists of four steps.

- First, we need to obtain an initial set. While the definitions of the *AppleReg*s are removed in the open-sourced XNU kernel code, we can still extract the mnemonic and configurations of *AppleReg* from the kernel binary. For instance, in the test function `arm64_ropjop_test`, the mnemonic of PA-related *AppleReg* can be found in string messages(e.g., `apsts` and `apctl`). We identify the PA-related *AppleReg*s based on these messages and gather an initial set of PA-related *AppleReg*s.
- Our next step is identifying their *alias registers* for building a complete system register set. System register redirection allows the XNU kernel to access registers belonging to the VM or the host OS using `_EL1` encoding (§2.2). Since the string message and test functions we used to create the initial set are accessible in either VM or host OS. Only the `_EL1` encoding can be used on both EL1 and EL2. Most of the *AppleReg*s in the initial set are `_EL1` encoding. To identify alias registers (e.g., `_EL12` encoding), we set a system register in the *m1n1+m1n1* environment to a flag value using `_EL1` encoding on EL1. We then read all register encodings on EL2. The `_EL12` encoding that contains the same flag value is the alias of that system register. The `_EL2` encodings are identified using the same method.
- Based on the alias registers identified in the previous step, we can then locate the code in binary that uses these alias registers and mark the *AppleReg*s in the same basic block as potential PA-related. If new PA-related *AppleReg*s are found, we repeat the second step and try to identify more PA-related *AppleReg*s.
- Finally, by collecting all *AppleReg*s in the XNU kernel binary and observing whether setting the encoding affects the behavior of PA instructions, we can verify that our findings contain all possible PA-related *AppleRegs*.

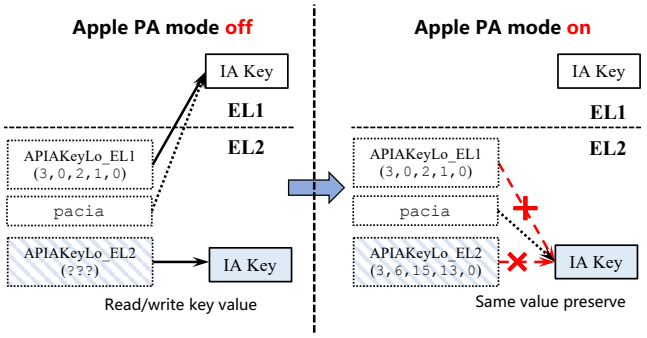


Figure 3: Controlling EL2 PA keys.

3.5 Reading/Writing Actual PA Key Values

On the M1 chip, PA registers are present in both EL1 and EL2. However, when attempting to read or write (using `msr` or `mrs` instructions) PA key registers with the `_EL1` encoding on M1 after enabling Apple PA, an exception is raised. Moreover, reading the values stored in PA key registers is critical for understanding how Apple customized its PA implementation. To achieve RC2, we propose two methods to enable the read and write operations on PA key registers, bypassing Apple's hardware protection on PA key registers.

3.5.1 Reading/Writing EL1 Actual Key Values

Although we can not access the PA key register directly, we find that the alias registers of EL1 PA key registers are readable/writable from EL2. Hence, we can first identify `_EL12` encodings of EL1 PA key registers using the technique in §3.4. With the `_EL12` encoding, we can read/write the EL1 PA key register from EL2, which bypasses the hardware protection.

3.5.2 Reading/Writing EL2 Actual Key Values

The macOS on M1 uses `_EL1` encoding to access PA key registers, such as using `APIAKeyLo_EL1` encoding `(3,0,2,1,0)` for accessing `APIA` key. However, when running in EL2, the VHE (discussed in §2.2) usually redirects the `_EL1` encoding accesses to the corresponding EL2 registers. As a result, we must first find the PA key register redirection targets and then design a technique to read/write corresponding targets.

PA key register redirection. We design experiments on *m1n1+Linux* environment to find the redirection targets of `_EL1` encoding when running in EL2 with Apple PA enabled. Our results show that the PA-related *AppleReg* (identified as `APCTL_EL1`) controls the redirection. As shown in Figure 3, when Apple PA is off, the PA instruction `pacia` on EL2 uses `APIA` key in EL1. In contrast, it uses the `APIA` key in EL2 when

Apple PA is on. As a result, to control real PA key values, we need to design a technique to read/write PA key in EL2.

Enabling EL2 key register read/write. Reading/writing EL2 PA key registers faces several challenges. First, M1 hardware prevents reading/writing the EL2 PA key registers when Apple mode is on. Second, we cannot use the encoding from a higher exception level to read/write EL2 key registers, as M1 does not implement EL3 exception level [13].

To overcome the above challenges, our key observation is PA key register values don't change when enabling Apple PA. In other words, if we manage to write a value to EL2 PA key register before enabling Apple mode, its value will be preserved and used after Apple mode is enabled. Using the same example in Figure 3, when Apple PA is off, we use the encoding of `APIAKeyLo_EL2` to read/write the EL2 `APIA` key. Its value is preserved and used by the `pacia` instruction when Apple PA is enabled. Note that the encoding of `APIAKeyLo_EL2` is unknown because the redirection of `APIAKeyLo_EL1` to the EL2 `APIA` key is only available when Apple PA is enabled, and we can not read the EL2 `APIA` key via `APIAKeyLo_EL1` encoding due to the hardware protection, the method used to identify `_EL2` encoding in §3.4 can not be used to identify `APIAKeyLo_EL2`. To get its encoding, we first get an over-approximate encoding set by collecting all *AppleReg*s that become non-readable after enabling the Apple PA. Then we identify the `_EL2` encoding of PA key registers by setting these *AppleReg*s and observing the results of `pac` instructions. With the identified `_EL2` encoding, we can use it to read/write the EL2 key registers before enabling Apple PA.

Summary. We can read the actual key values of EL1 (using §3.5.1). However, it is not possible to read the actual key values of EL2 using the same method due to the absence of EL3 on the M1 chip. To address this issue, we propose reading and writing the EL2 PA key value before enabling Apple PA (§3.5.2). Although reading EL2 key values is infeasible after enabling Apple PA, writing PA keys in advance is sufficient for us to investigate the behavior of PA instructions on EL2.

3.6 PA Instruction Behavior Profiling

Based on identified PA-related *AppleReg*s (§3.4) and the ability to read/write PA keys (§3.5), we propose a method that can effectively analyze the interplay between PA system registers (including PA control registers and PA key registers) and PA instructions. Here, we use the controlled variables method [17] in our experiments. More specifically, we change one bit of the PA control registers at a time, set PA key registers using fixed values, and observe the PA key value changes and PA instruction output to understand the controllability of PA control registers. We can deduce the interplay between PA system registers and PA instruction behavior. We perform the following steps to profile PA instructions.

First, we extract the typical PA system register values from *m1n1+macOS* environment. Next, we change one bit and set the value to the actual register. We then read the PA key on EL1 and EL2 using techniques in §3.5. Finally, we need to compare the results of cross-EL (user mode EL0 vs. kernel mode EL1), cross-key (different PA keys: `APIA/APDA/APIB/APDB/APGA`), cross-VM (virtual machine (EL1, `HCR_EL2.TGE=0`) and the host OS (EL2, `HCR_EL2.TGE=1`)), cross-boot (different reboot rounds). It is worth noting that the single-variable principle should be followed each time the control and key registers are set. By establishing the connection between the register (PA-related control register and key register) settings and the results of the instructions (key access instructions and pac instructions), we can reveal the hardware implementation of Apple PA.

To address the potential explosion, we mark specific bits in PA control registers as *valid control bits*. Leveraging known information of certain bits, such as the Apple PA enable bit [14] and the four control bits specified by the ARM specification [8], the rest are marked as *unknown bits*. Different combinations of valid control bits indicate different states. We identify new valid control bits by observing the impact of setting unknown bits in different states on PA instruction behavior. This probing process continues until no further valid control bits are discovered. We identify a total of nine valid control bits, of which eight are utilized in the XNU kernel. Thus, during the profiling of PA instruction behavior, we only need to consider the influence of these nine valid control bits.

3.7 Hypervisor-based Kernel Dynamic Analysis

To achieve RC4, we develop a kernel dynamic analysis system based on *m1n1+macOS*. Currently, the only kernel debugging framework for ARM-based XNU kernel on M1 is LLDB [33] provided by Apple. However, it does not meet our requirements because the LLDB can only be used after kernel initialization [36]. Moreover, the LLDB for Apple Silicon does not support active kernel debugging (i.e., breakpoints, single-step debugging) [19].

We replace instructions of macOS (EL1) with traps (`hvc` instruction) and handle the traps in m1n1 (EL2). As a result, the m1n1 running in the hypervisor mode allows us to boot the macOS and trace the boot-up. Moreover, after macOS booted, our framework can suspend its execution and modify CPU registers and memory at any time.

4 Revealing PA Hardware Implementation on M1

In this section, we first give an overview of our findings on Apple PA customization. We then discuss how Apple PA mitigates cross-domain attacks in detail. The findings on PA registers and PAC algorithm are based on techniques in §3, we leave the experimental details out for brevity. For the cross-domain mitigations, we give detailed experiment settings.

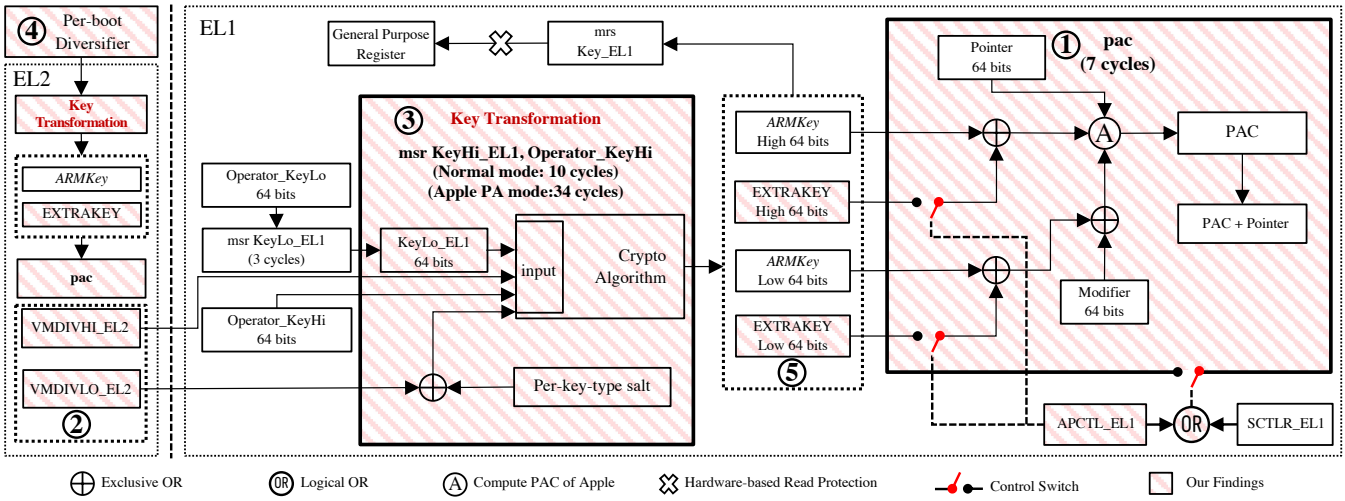


Figure 4: Findings Overview of Apple PA. Apple PA introduces key transformation, new PA-related registers, hardware-based read protection, and customized `pac` instructions.

4.1 Finding Overview

Our new findings are summarized in Figure 4, including ① Apple's customization on PA control registers, controllability, and PAC algorithm; ②-⑤ Hardware-based cross-domain attack mitigations.

1) New registers. We found that Apple M1 introduces two new PA control registers. `EXTRAKEY_EL1` is used for differentiating the PAC computation between user and kernel space, while `VMDIV_EL2` is for diversifying between the host and virtual machines. In addition, we identify the `_EL2` encodings of PA key registers and find that the redirection of `_EL1` to the EL2 key register is only available when Apple PA is enabled.

2) Controllability. Apple uses `APCTL_EL1` as the main PA control register. `APCTL_EL1` has five bits for controlling Apple PA. AsahiLinux found that `bit[0]` is the switch used to enable Apple PA. Besides these existing findings, we identify the controllability of `bits[1-4]` of `APCTL_EL1`.

Based on the technique in §3.6, we find that the `bit[2]` of `APCTL_EL1` is used to enable PA in user space, and `bit[3]` is used for kernel space. We also find that the two bits can be used together with original ARM per-key-type switches (`EnIA/IB/DA/DB` bits of `SCTLR_EL1`). Specifically, PA in an EL using a key is enabled if any corresponding bit in `APCTL_EL1` and in `SCTLR_EL1` is set. In addition, we find that the `bit[1]` and `bit[4]` control the `EXTRAKEY_EL1` (discuss in §4.2.5).

3) PAC algorithm. For the PAC algorithm, we run `pac` instructions on QEMU (uses QARMA [15, 38]) and Apple M1. The results show that Apple implements a customized algorithm that is different from QARMA. Moreover, we swap the values between the inputs of `pac` instruction and observe the PAC. The result shows that PAC algorithm on M1 takes the XOR result of lower 64 bits of the key and modifier as input.

4) Cross-domain attack mitigation. One main contribution of our paper is that we revealed how Apple customizes the PA hardware to mitigate cross-domain attacks (defined in §4.2.1). Our experiments reveal that Apple PA uses an `VMDIV_EL2` register to diversify the PAC computation between the host OS and VMs (② in Figure 4); Apple PA conducts a per-key-type key transformation to differentiate the PAC computation using different keys (③); Apple PA introduces a per-boot diversifier to diversify the PAC computation between different CPU boots (④); Apple PA uses a `EXTRAKEY_EL1` register to differentiate the PAC computation between user mode and kernel mode (⑤). Details are presented in the following section.

4.2 Cross-domain Attack Mitigation

We first give a formal definition of cross-domain attack and then present our experiments and findings on Apple M1 for each type of cross-domain attack.

4.2.1 Definition of Cross-domain Attack

ARM pointer authentication is vulnerable to pointer substitution attacks by design. To substitute kernel pointers, one common way is to sign a pointer in user space and use it to replace the signed kernel pointer. This attack replaces pointers of different exception levels (user vs kernel) and thus is termed as *cross-EL* attack. Similarly, the attacker can replace pointers between the host and virtual machine, leading to *cross-VM* attack. Moreover, the attacker can replace pointers signed by different keys, leading to *cross-key* attack. In addition, the attacker can replace pointers across different system boots, leading to *cross-boot* attack. In this paper, we generalize these four types of pointer substitution attacks as *cross-domain* attack.

We define domains formally in the following.

Definition 1 - Domains. $\mathcal{D} = \{v \in VM, k \in Key, b \in Boot, e \in EL \mid (v,k,b,e)\}$ denotes the set of all domains. Each element $d \in \mathcal{D}$ denotes a unique domain instance and is a quadruple. Items in the quadruple are from four different sets, where:

- $VM = \{0, 1, 2 \ldots\}$. VM denotes the set of the Host instance and all VM instances. For ease of representation, we use

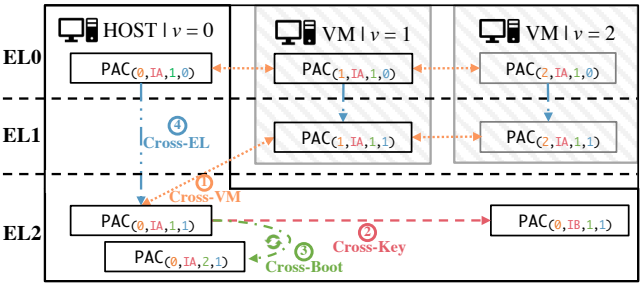


Figure 5: Cross-domain attacks. (Inputs $(\text{ptr}, \text{mod}, \text{keyval})$ used for PAC computation are the same in all domains.)

0 to denote the Host instance and other numbers to denote VM instances.

- $Key = \{\text{IA}, \text{IB}, \text{DA}, \text{DA}, \text{GA}\}$. *Key* denotes the set of different PA keys that can be used for PAC computation.
- $Boot = \{1, 2, 3 \ldots\}$. *Boot* denotes the set of numbers of boot rounds.
- $EL = \{0, 1\}$. *EL* denotes the set of exception levels of the machine. We use 0 to denote user and 1 for kernel.

The result of PAC computation should be bound to the affiliated domain to enforce the least privilege principle. We have the definition of the domain PAC computation.

Definition 2 - Domain PAC computation. In a domain d,

$$PAC_d = \text{HMAC}_d(ptr, mod, keyval),$$

ptr denotes the pointer to be signed, *mod* denotes the modifier for PAC computation, and *keyval* is the key value used in the HMAC algorithm.

Definition 3 - Cross-Domain attacks. Let d_a be the domain of the attacker, and d_v be the domain of the victim. The cross-domain attack happens when an attacker can use a signed pointer in d_a to substitute a signed pointer in d_v. Corresponding to four types of domains, there are four types of cross-domain attacks against ARM PA.

Definition 3.1 - Cross-VM attacks. $d_a = (v_a, k, b, e) \in \mathcal{D}, d_v = (v_v, k, b, e) \in \mathcal{D}, v_a \neq v_v, \exists input_a = input_v = (ptr, mod, keyval)$, such that $PAC_{d_a} = PAC_{d_v}$.

An attacker can launch cross-VM attacks from a different virtual machine or the host OS v_a from the victim v_v by signing the same pointers with the victim. Cross-VM attacks require the same configuration of PA hardware (registers, instructions) to be used in different v_a and v_v to compute the PAC. Although the attacker and the victim are in different virtual machines or host OS. However, ARM PA does not support VHE (§2.2), and all virtual machines and host OS share the same PA hardware implementation. The requirements of the cross-VM attacks can be easily satisfied.

Cross-VM attacks can be divided into three types according to the values of v_a and v_v. The first one is a VM-Host attack, where the attacker tries to compromise the PAC in the host OS from a virtual machine ($v_a > 0, v_v = 0$). The second is VM-VM attacks, where the attacker tries to attack another virtual machine from a virtual machine ($v_a > 0, v_v > 0$). The third one is Host-VM attacks, where the attacker tries to compromise the PAC in a virtual machine from the host OS ($v_a = 0, v_v > 0$). As shown by ① in Figure 5, when the inputs of the PAC computation are equal, $PAC_{(0,IA,1,1)}$, $PAC_{(1,IA,1,1)}$ and $PAC_{(2,IA,1,1)}$ are equal, then the above three types of attacks are feasible.

Definition 3.2 - Cross-Key attacks. $d_a = (v, k_a, b, e) \in \mathcal{D}, d_v = (v, k_v, b, e) \in \mathcal{D}, k_a \neq k_v, \exists input_a = input_v = (ptr, mod, keyval)$, such that $PAC_{d_a} = PAC_{d_v}$.

An attacker can launch cross-key attacks by using a different key k_a with the victim key k_v to sign the same pointer. Cross-key attacks require that k_a and k_v should hold the same key value. The requirement can be easily achieved when a process uses a constant initializer to initialize all the keys. As shown by ② in Figure 5, when the inputs of the PAC computation are equal, if the $PAC_{(0,IA,1,1)}$ using `APIA` key is equal to $PAC_{(0,IB,1,1)}$ using `APIB` key, the attacker can forge $PAC_{(0,IA,1,1)}$ to substitute $PAC_{(0,IB,1,1)}$ to bypass the PA protection to achieve further attack such as control flow hijack.

Definition 3.3 - Cross-Boot attacks. $d_a = (v, k, b_a, e) \in \mathcal{D}, d_v = (v, k, b_v, e) \in \mathcal{D}, b_a < b_v, \exists input_a = input_v = (ptr, mod, keyval)$, such that $PAC_{d_a} = PAC_{d_v}$.

In the cross-boot attack, the attacker at boot round b_a attempts to infer the PAC of the same pointer in the later boot round b_v. Cross-boot attacks require that the PA keys should hold the same key value between different CPU boot rounds. These requirements can be easily achieved when all keys are initialized with fixed constants after each CPU boots.

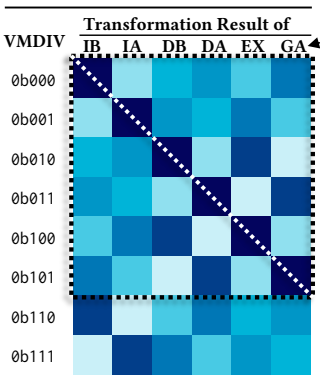
As shown by ③ in Figure 5, when the inputs of PAC computation are equal, if the $PAC_{(0,IA,1,1)}$ calculated in boot round 1 is equal to $PAC_{(0,IA,2,1)}$ calculated in boot round 2, the attacker can forge $PAC_{(0,IA,1,1)}$ to substitute $PAC_{(0,IA,2,1)}$ to bypass the PA protection. Moreover, the attacker can reuse $PAC_{(0,IA,1,1)}$ to substitute the PAC of any other boot rounds, leading to a continuous PA protection bypass.

Definition 3.4 - Cross-EL attacks. $d_a = (v, k, b, e_a) \in \mathcal{D}, d_v = (v, k, b, e_v) \in \mathcal{D}, e_a = 0, e_v = 1, \exists input_a = input_v = (ptr, mod, keyval)$, such that $PAC_{d_a} = PAC_{d_v}$.

Cross-EL attacks require that the used PA keys hold the same values across the user mode and the kernel mode. The requirement can be achieved when the OS does not reconfigure PA keys during switching between the user mode and kernel mode. As shown by ④ in Figure 5, when inputs of the PAC computation are equal, if the $PAC_{(0,IA,1,0)}$ calculated in the user mode is equal to the $PAC_{(0,IA,1,1)}$ of the kernel mode, the attacker can forge $PAC_{(0,IA,1,0)}$ to substitute $PAC_{(0,IA,1,1)}$ to bypass the PA protection of kernel mode to achieve further attack such as kernel control flow hijack.

4.2.2 Cross-VM Mitigation

Confirming M1 has cross-VM mitigation. We run the macOS in EL2 as the *host OS* and run multiple virtual ma-



(a) Each **cell color represents a key transformation result** when setting IB, IA, DB, DA, EXTRA KEY, and GA to 0 at EL1 with different VMDIV. It shows apparent symmetry in the dashed box, and only XOR matches this symmetry.

Per-key-type Salt of					
IB	IA	DB	DA	EX	GA
0	1	2	3	4	5
1	0	2	3	4	5
2	3	0	1	6	7
3	2	1	0	7	6
4	5	6	7	0	1
5	4	7	6	1	0
6	7	4	5	2	3
7	6	5	4	3	2

(b) Only 8 combinations of *per-key-type salts* that XORed with VMDIV will produce the same symmetry, which is why Apple PA mitigates the cross-key attack. With *per-key-type salts*, the exact value of the key will be transformed into different ones.

Figure 6: Key transformation results and the possible combinations of *per-key-type salt*s (raw data in Appendix Table 6).

chines (VM) on top of it. We set the same value to PA keys via `Key_EL1` encodings in host OS and VMs. Then we sign the same pointer with the same key and modifier on host OS and VMs. As shown in Figure 5, the PAC computations are different between different VMs and host OS ($PAC_{(1,IA,1,1)} \neq PAC_{(0,IA,1,1)} \neq PAC_{(2,IA,1,1)}$). Since XNU kernel does not reconfigure the PA key during VM-Host switch, and all host OS and VMs use the same fixed value to initialize PA keys, we know that M1 has cross-VM mitigation.

Revealing how M1 achieves cross-VM mitigation. Based on the profiling method (§3.6), we find that when the Apple PA is enabled, setting PA key using `msr KeyHi_EL1, operator` instruction introduces a key transformation in VM. Meanwhile, the key transformation is also deployed on host OS.

In experiments, we first set `VMDIV_EL2` to different values, as shown in Figure 6. We then use the `msr Key_EL1, operator` instruction in VM to set the same value for the PA key. The experimental results show that the key transformation of VM uses `VMDIV_EL2` as its random source. The key transformation is also deployed for the host OS, but with a different random source, leading to different key transformations for the VM and host OS.

> **Finding 1.** Writing high 64 bits of a PA key triggers a key transformation process on the whole 128-bit key. On VMs, the key transformation uses a per-VM diversifier `VMDIV_EL2` to defend VM-VM attacks. On host OS, the key transformation uses a different diversifier rather than `VMDIV_EL2` to defend VM-Host attacks.

Conclusion. In the XNU kernel, `VMDIV_EL2` (② in Figure 4) is set to a per-VM value and the random sources of key transformation are different between EL1 (VM) and EL2 (host OS). For different VMs and host OS, even if the same value is set to PA key using the `msr key_EL1, operator` instruction, the actual key values for PAC computation are different.

Limitation discussion. As we mentioned in **Definition 3.1**, there are three types of cross-VM attacks: VM-Host, VM-VM, and Host-VM attacks. The former two can be defeated by current M1 PA hardware. However, the Host-VM attack can be successful on M1. As a result, if the host OS can leak the value of the PA key used by the VM, then the attacker can infer the PAC computed in the VM at the host OS.

To protect VMs, we suggest introducing a switch that only the VM can control whether the host OS can access the actual key value of the VM. Also, the VM should avoid using fixed value as the input of key transformation.

4.2.3 Cross-Key Mitigation

Confirming M1 has cross-Key mitigation. After Apple PA initialization, we set the same value for different PA keys via `_EL1` encodings. We then sign the same pointer with the same modifier using different keys. The results are not the same, confirming that M1 has cross-key mitigation.

Revealing how M1 achieves cross-key mitigation. Our key finding is that a per-key-type salt is introduced in the key transformation process. To find out how the salt is used in key transformation, we keep the value of the key the same, altering only the value of `VMDIV_EL2` and observing the corresponding actual key value for the five different keys. As shown in Figure 6, The results show that six different hard-coded constants are introduced in the key transformation for different key types. We use the term *per-key-type salt* to represent these constants. In addition, the XOR result of *per-key-type salt* and `VMDIV_EL2` is one of the inputs for the key transformation on EL1. There are eight possible combinations for the lowest three bits of *per-key-type salt*s.

> **Finding 2.** The key transformation introduces a *per-key-type salt* to defend cross-key attacks. The XOR result of *per-key-type salt* and the diversifier is one of the inputs for key transformation.

Conclusion. Due to the presence of per-key-type salt (③ in Figure 4), the result of key transformation is different for different key types. So when Apple PA is enabled, even if setting different PA key registers to the same value, the actual key value used for PAC computation is different, achieving cross-key mitigation.

4.2.4 Cross-Boot Mitigation

Confirming M1 has cross-boot mitigation. The XNU kernel uses the `msr Key_EL1, operator` instruction to set the PA key to a fixed value during initialization. After each reboot, we sign the same pointer using the same modifier and PA key. The results differ after each reboot, indicating that M1 has cross-boot mitigation.

Revealing how M1 achieves cross-boot mitigation. We use the `msr KeyLo/Hi_EL1, operator` instruction on the host OS (running in EL2) to set the PA key registers. Combined with the reboot operation and observing the results of `pac` instructions, the results show that the host OS deploys key transformation, which uses a per-boot diversifier as one of its inputs.

> **Finding 3.** The key transformation on EL2 uses a per-boot diversifier as one of its inputs, mitigating cross-boot attacks.

Conclusion. Since the host OS uses a per-boot diversifier for key transformation(④ in Figure 4), even if the XNU kernel uses hardcoded values to set the PA key registers, the actual key value is different between system boots. This design effectively mitigates the cross-boot attack.

4.2.5 Cross-EL Mitigation

Confirming M1 has cross-EL mitigation. On macOS, we use the same key and modifier to sign the same pointer in both user mode and kernel mode. The results are different, showing that M1 has cross-EL mitigation.

Revealing how M1 achieves cross-EL mitigation. Apple PA introduces a new 128-bit register (`EXTRAKEY_EL1`) to differentiate PAC between user and kernel modes. In the experiments, we first set up the `EXTRAKEY_EL1` and `APCTL_EL1`. Specifically, we set the *ARMKey*s and `EXTRAKEY_EL1` directly without triggering key transformation and observe the results of `pac` instructions. The results show that `EXTRAKEY_EL1` is used to XOR with *ARMKey*s and the XOR result is the actual PA key value used for PAC computation. Moreover, the `bit[4]` and `bit[1]` of `APCTL_EL1` are used to enable the `EXTRAKEY_EL1` on user mode or kernel mode.

> **Finding 4.** Apple PA differentiates the PAC computation between user space and kernel space by XORing *ARMKeys* with `EXTRAKEY_EL1` on user space. The XOR result will be the actual key value for PAC computation.

Conclusion. The XNU kernel only enables `EXTRAKEY_EL1` (⑤ in Figure 4) in the user mode. The results of PAC computation in the user mode and kernel mode are different even when the inputs of `pac` instruction are the same. This design mitigates the cross-EL attack.

5 Analyzing PA Usage in XNU Kernel

macOS on M1 heavily relies on PA to enforce control-flow integrity (CFI) and data-flow integrity (DFI). The typical usage of PA includes the sign/auth interface usage and key management. Therefore, we conduct an analysis of both of them, as presented in the following sections.

Table 1: Results of signing interface analysis. (DB key is not used in XNU kernel.)

Key Usage	Modifier	Target
IA (762376)	Hash(function_type) (38564)	Function pointer
	Hash(function_name) + storage address(384)	Recovery Handler/ Corecrypto related
	Storage address (27568)	ppl_handler(89)
		Copy/destroy_helper_block (626)
		Block_invoke function(115)
		Ptrs in seg: __auth_ptr (25452)
		Block function pointer (1286)
	Hash(root_class, function_type, function_name) + Storage address (694596)	Vtable entry(694596)
	Zero (1264)	__chkstk_darwin* func (156)/ BluetoothFamily function (9)/ kext_weak_symbol_referenced/ Parameter func ptr (1098)
DA (31225)	Hash(data_field_type) + Storage addr (2645)	Proc0 (1)
		__NSConcreteGlobalBlock(115)
		_Block_descriptor (115)
		sysctl_oid_list (2414)
	Hash(data_pointer_name)+ Storage addr (464)	Data pointer
	Hash(root_class) + Storage addr (28116)	V-table pointer
IB(110852)	SP	Return address
DB	-	-
GA	Storage Address	Thread state/ Exception state/ Data Blob

5.1 Sign/Auth Interfaces Analysis

The PA sign/auth results are mainly decided by three factors: the target pointer, the modifier, and the key. To realize the analysis, we first need to analyze the values of the targets and the modifiers under complex data propagation. Particularly, we use the hypervisor-based dynamic analysis (§3.7) to record operators with complex data propagation of sign/auth interfaces. Second, we need to reveal the high-level policies behind the undisclosed constants used in modifiers. We denote these constants as modifier constants (MCs). To resolve this problem, we follow the single variable principle to find out which kind of static information decides the MCs among all possible ones. Specifically, the possible types of static information for function pointers can be function names, member field names, function types (including the number, sequence, and types of parameters and return values [32]), and the root class (for vfunc pointers) can be used to generate MCs. The static information for data pointers can be the data type and member field names. Based on these two methods, we analyze three factors for all sign/auth interfaces in the XNU kernel.

For the pointer target, we find that Apple protects function pointers, vtable pointers, and vtable entries based on PA. However, for other sensitive data, Apple uses highly customized policies to decide the protection targets, which gives rise to potential attack chances. For example, there have been real attacks [1,20] that bypass PA protection based on unprotected sensitive data pointers.

Table 2: Scope of each key. G: global; P: per-process; V: per-VM; EX: EXTRAKEY; VK: VMKEY.

Key	IA	DA	IB	DB	GA	EX	VK
Scope	G	G	P	P	G	P	V

Table 3: Scope of each PAC instruction. U: user space; K: kernel space; e: arm64e; ne: non-arm64e G: global; P: per-process; -: unavailable.

	pacia	pacda	pacib	pacdb	pacga
U(e)	P	P	P	P	P
U(ne)	-	-	P	-	P
K	G	G	P	P	G

Finding 5. XNU kernel uses nine types of signing modifiers and six policies for generating MCs in the latest XNU kernel.

For the signing modifier, we find that Apple uses nine types of signing modifiers and six policies for generating MCs in the XNU kernel (only five types of signing modifiers and two types of MCs are mentioned in official documentation [3,28]). However, we find that many signing targets still share the same signing modifier. Real attacks [16, 23, 24] demonstrate that this problem may lead to practical substitution attacks.

For the key usage, we find that Apple uses IB key to sign return addresses and uses IA key for all other function pointers. Apple uses DA key to sign data pointers and leaves DB key for user-space programs. Apple uses GA key to sign interrupt context and other sensitive data blobs.

5.2 Key Management

For the key management in the XNU kernel, the only available information is that the IB key value is per-process as disclosed by Apple [28]. Previous researchers were unable to properly analyze key management because of Apple's undisclosed customization on PA hardware. Therefore, after revealing Apple's PA hardware implementation, we analyze the key management in XNU kernel.

We list the configuration of APIA/DA/IB/DB/GA, EXTRAKEY, and VMKEY in Table 2. Moreover, XNU kernel disables EnIA/DA/DB bits of SCTLR_EL1 for non-arm64e [4] processes. Combining the operations above, we summarize our findings of PAC instruction scope in Table 3.

1) Fine-grained key management. XNU kernel only enables EXTRAKEY on user space. Meanwhile, the kernel sets EXTRAKEY to per-process values and APIA/DA/GA to static values. Based on the findings of EXTRAKEY (§4.2.5), the actual key values for pacia/da/ga instructions are per-process in user space and global in kernel space.

2) Process-dependent controllability. All pac instructions are still available in kernel space without enabling bits in SCTLR_EL1 because bit[3] (kernel PAC switch) of APCTL_EL1 is always enabled in XNU kernel. Therefore, the kernel can choose whether to enable PAC computation of user mode by setting per-key-type switches in SCTLR_EL1. For example, XNU kernel disables EnIA/DA/DB bits of SCTLR_EL1 for non-arm64e processes, so these processes can not use pacia/pacda/pacdb instructions to generate PAC in user mode.

6 Security Analysis

6.1 Threat Model

We assume that the attacker has arbitrary kernel memory read/write capability by exploiting known CVEs. We also assume that the attacker can interrupt the XNU kernel at anytime. The attacker aims to corrupt the protected function/data pointers to break the PA-based protection to launch control-flow or data-flow hijack attacks. This threat model is practical as the design goal of PA is to protect pointers under arbitrary memory read/write. Besides, we also assume the existing defense mechanisms on Apple M1 are all enabled, including secure boot, stack protection, DEP, KASLR, and Apple-specific protection [45]. We assume that MMU cannot be disabled and the attacker can only access memory using virtual addresses. We do not consider side-channel attacks, such as PACMAN attack [37], and hardware attacks, such as Rowhammer [27].

6.2 Attack Surface Analysis

We identify four attack surfaces (AS❶-❹). We further implement our analysis tool based on IDAPython and CodeQL to analyze all of them. More specifically, we implement an intra-procedural data-flow analysis (identifying AS❶, AS❷, and AS❸) as well as other binary analysis based on IDAPython. For identifying the data propagation from non-sensitive data propagation to sensitive data pointer in AS❶, we implement an inter-procedural data-flow analysis based on CodeQL.

6.2.1 Incomplete Sensitive Data Identification (AS❶)

For PA-based CFI/DFI, sensitive data includes all data that can affect control and sensitive data flow. If PA-based CFI/DFI implementation does not protect all sensitive data, there will be unprotected sensitive data in memory. This implementation flaw can lead to an attacker bypassing the PA-based CFI/DFI by modifying the unprotected sensitive data in memory.

Analysis method. As shown in Listing 1, if sensitive data is loaded from memory (Line 2, 7, and 11) and signed later (Line 5 and 8) or influences the control flow (Line 12) without authentication, it means that the sensitive data is unprotected. We implement an *intra-procedural data-flow analysis* to identify these unprotected sensitive data. Besides, it is worth noting that Apple marks the data pointer as sensitive by annotating it using Apple's customized language extension. However, due to the complex data-flow propagation, some unprotected non-sensitive data in memory will propagate to sensitive data pointers. An attacker can modify the sensitive data pointers indirectly to bypass the PA-based kernel DFI. We implement

an *inter-procedural data-flow analysis* to identify such data propagation.

6.2.2 Incomplete Interrupt Context Protection (AS❷)

When the interrupt is enabled and an exception is triggered, the contents of registers, namely the interrupt context, will be spilled in memory. The interrupt context may contain unprotected sensitive data such as function pointers. An attacker can bypass PA-based CFI/DFI by modifying sensitive data spilled into memory due to interrupts.

Analysis method. XNU kernel uses `sign_thread_state` to sign the exception state. More specifically, the `sign_thread_state` function signs the `{pc, cpsr, lr, x16, x17}` of interrupt context. These five registers are interrupt-safe registers, while the others are interrupt-unsafe registers. Sensitive data should not be stored in interrupt-unsafe registers when the interrupt is enabled. Similarly, to identify AS❷, we identify which sensitive data are stored in interrupt-unsafe registers by binary analysis. In addition, because some sensitive data has to be passed into the interrupt-unsafe register, M1 prevents these data from being spilled into memory by disabling the interrupt. We also identify if sensitive data is propagated into interrupt-unsafe registers before the interrupt.

6.2.3 Signing Gadget (AS❸)

A signing gadget is a code gadget that calls the signing interface. For an attacker with a signing gadget, the PA protection can be bypassed directly with the help of the signing gadget when the attacker can control the inputs of the signing gadget, or the attacker can use the signing gadget to get the signed target for conducting substitution attack and eventually bypass the PA protection. We classify the signing gadget by the signing target into function-level and instruction-level. An attacker with a function-level signing gadget can substitute a signed sensitive data structure to conduct further attacks, while an attacker with an instruction-level signing gadget can forge signed pointers to conduct pointer substitution attacks.

Analysis method. The exploitability of the signing gadget depends on whether multiple signing targets share the same signing interface. When sharing, attackers can leverage the signing gadget to sign multiple targets, enabling substitution attacks. To assess the exploitability of the signing gadget, we analyze the signing interfaces sharing in the XNU kernel. For the function-level signing gadget, we first collect all call sites of the function-level signing gadget based on binary static analysis and perform analysis on these call sites to determine if the function-level signing gadget is shared by multiple signing targets. For the instruction-level signing gadget, we collect all `pac` instructions and classify them based on the signing modifier value (as in §5.1).

6.2.4 Key Leakage (AS❹)

If the PA key value used to compute the PAC is stored directly in memory without being encrypted, an attacker can leak the key value to complete further attacks. For example, when a process-dependent PA-based CFI/DFI is implemented, an attacker can leak the PA key values of other processes and forge signed pointers based on the leaked PA key value to bypass the PA-based CFI/DFI.

```
1  ; pattern 1 - X8 is loaded from memory
2  ldrsw           x8, [x9,#0xC]!
3  add             x8, x8, x9
4  ...
5  pacia           x8, x9
6  ; pattern 2 - X8 is loaded from memory
7  ldr             x8, [sp,#0x30]
8  pacda           x8, x11
9  ; pattern 3 - X6 is loaded from memory
10 ; xnu-8019
11 ldr             x6, [x2]
12 and             x17, x6, #0xff
13 ldrsh           x17, [x16,x17,lsl#1]
14 add             x17, x16, x17
15 br              x17
16 ;xnu-7195    - sensitive data is stored in
   ↪ interrupt-unsafe register
17 ...
18 br              x20
```

Listing 1: Sensitive data are loaded without authentication.

Analysis method. For key leakage, we first reverse engineer the hardware implementation of Apple-specific PA key registers (§4) and then analyze the key management implementation in the XNU kernel (§5.2). Combining these findings, we perform a security analysis for unencrypted per-process PA keys stored in memory. Since these keys are not encrypted, an attacker can make multiple processes' per-process PA key values equal by leaking the key and modifying it. Our security analysis of key management is to analyze the security problems that may result when the per-process PA keys of different processes have equal values.

6.2.5 Analysis Results

We summarize our findings in Table 4. For AS❶, we identify 153 cases (xnu-8019) that can be used to bypass PA-based kernel CFI/DFI (29 for CFI, 124 for DFI). Meanwhile, we find that 52 out of 81 sensitive data pointers propagated from non-sensitive data (5 cases are propagated directly from non-sensitive data).

For AS❷, as shown in line 18 of Listing 1, we find 18 cases of `br` instruction using interrupt-unsafe register `x20` in xnu-7195. These cases are fixed in xnu-8019 (changing to `x17`). We also find hundreds of cases in xnu-8019, most of the cases are omitted from the earlier XNU kernel version. Besides, we identify 17 cases that sensitive data is propagated into interrupt-unsafe registers before disabling the interrupt.

For AS❸, since it is the most commonly exploited attack surface to bypass PA protection in recent years [16, 23, 24], Apple has improved its defense against this attack surface several times by removing the unnecessary signing interface, using PA to protect the data pointer and disabling the interrupt. However, we find that the mitigation of this attack surface is incomplete, and the inputs of `sign_thread_state` function could be spilled to memory, resulting that an attacker can modify the contents `{pc, cpsr, lr, x16, x17}` of a kernel

Table 4: Identified, validated, and fixed cases.

AS \ Result	Identified	Validated	Fixed Cases
❶	153	83	6
❷	17+18*	2	2+18*
❸	1	1	1
❹	2	2	-

* 18 cases are identified in XNU-7195, and all of them are fixed in XNU-8019.

exception state to achieve arbitrary kernel function call.

For AS❹, we find that XNU kernel assigns `APIB` and `EXTRAKEY` key values for each process (`jop_pid` field for `EXTRAKEY`, `rop_pid` field for `APIB/APDB`) and that the key values are stored in memory without encryption. For `APIB`, it is used to sign the kernel return address while using `pacibsp` (signs `LR` with `SP` as the modifier). However, the `pacibsp` instruction could be vulnerable to substitution attacks because different return addresses may be signed with the same `SP` [30]. For `EXTRAKEY`, based on findings in §5.2, if two processes share the same `EXTRAKEY` value, the result of `pacia`, `pacda`, and `pacga` on user mode with the same inputs will be the same. An attacker can replace the `jop_pid` to forge a pointer to hijack the user-space control/data flow of other processes.

6.3 Result Validation

Validation method. We validate our above findings on the XNU kernel (xnu-8019) by simulating the attacker's *arbitrary kernel memory* read/write at *arbitrary moment* capability based on our dynamic analysis framework (§3.7).

More specifically, we specify the arbitrary moment by setting breakpoints. After trapping into EL2, the `HCR_EL2.TGE` will initially be zero [6]. At this moment, using the `s1e1w` instruction [7] can check if a virtual memory location is writable for access from EL1 based on the stage-1 page table. At the same time, we can also determine whether the interrupt in EL1 is enabled based on the value of `SPSR_EL2`. If the virtual memory location is writable, we can modify the contents of the memory as the attacker and return to the virtual machine. The macOS kernel log can be read through the serial port to confirm the validation result. Reasons for validation failure include: memory location is read-only, the interrupt is disabled, and difficult to trigger (complex call paths).

Validation results. As shown in Table 4, we validate that 83 cases of AS❶ and 2 cases of AS❷ are exploitable (xnu-8019). For AS❸, the validation results show that Apple's previous fix is still vulnerable to modification of `sign_thread_state`'s spilled inputs. We modify a kernel exception state's spilled `PC` and achieve an arbitrary kernel function call. Similar signing gadgets have been exploited to bypass PA protection [16, 23] in the XNU kernel. These actual attacks cross-validate the exploitability of this case. For AS❹, we validate two attacks against `APIB` and `EXTRAKEY` keys. The results show that an attacker can bypass PA protection by leaking process-dependent keys and modifying them to make them equal between different processes. Similar techniques were exploited in the real attack [20], which cross-validates our findings.

Case study. As *pattern 3* shown in Listing 1, sensitive data is loaded into `x6` without authentication at line 11 and used as an index to load data into `x17` at line 13. The content of `x17` will then be used as an offset value for an indirect call at line 15. We validate that the memory address used at line 11 is writable. An attacker can craft the sensitive data loaded from memory at lines 11 and 13 to achieve an arbitrary kernel function call.

Although Apple prevents the `sign_thread_state` calls for the user thread state from being reused to sign the kernel exception state by utilizing PA to protect the user thread state pointer and disabling the interrupt. However, we find that the inputs (`x0`: thread state pointer, `x1`: pc, `x2`: cpsr, `x3`: lr, `x4`: x16, `x5`: x17) of `sign_thread_state` function could still be spilled into memory. As a result, the attacker can modify `x0` to a kernel exception state and `x1` to a kernel function, and finally implement an arbitrary kernel function call.

6.4 Apple's Response

We report all our findings to Apple teams and have a nine-month ongoing communication with them. Based on our communication, we divide our findings into two categories.

1) Fixed. For AS❶, we identified 6 cases in xnu-8019 that have been fixed in the latest XNU kernel. For AS❷, Apple has addressed this issue in a security update and acknowledged our contribution in the security advisory. For AS❸, Apple has fixed the identified vulnerability in the latest releases and assigned us CVE-2023-32424.

2) Potential enhancements. Apple considers these findings as potential enhancements. In response to our findings in AS❶ and AS❹, Apple security team considers these cases as no-need-to-fix as there are no suitable vulnerabilities to trigger these cases. They consider these cases as potential enhancement points in their future releases.

6.5 Mitigation Discussion

Apple has been improving XNU kernel to mitigate PA attacks. However, as indicated by our security analysis, it is still practical to launch PA attacks. In the following, we first compare PA adoption evolution across all XNU versions on M1 and then give mitigation suggestions.

6.5.1 Evolution of PA Protection

We summarize the evolution of PA protection in Table 5.

1) Increasing protection targets (for AS❶). Apple protects more sensitive data gradually using PA. However, it is worth noting that the number of sensitive data pointers changes from version to version, which means there are still unprotected sensitive pointers in the XNU kernel [20].

2) Improving interrupt protection (for AS❷). Apple tries to avoid storing sensitive data in the interrupt-unsafe register

Table 5: Comparison of PA across XNU kernel versions.

Changes of PA Protection*		xnu-7195	xnu-8019	xnu-8020	xnu-8792
AS❶	Data blob	✗	✔	✔	✔
	Null pointer	✗	✔	✔	✔
	Corecrypto-related function pointer	✗	✔	✔	✔
	Sensitive data field	86	81	87	82
AS❷	Interrupt disabling before signing	✗	✔	✔	✔
AS❸	Number of protected thread state types	3	2	2	2
	Recovery handler protection	✔	✔	✔	✗

* AS ❹ is the same across XNU versions.

when the interrupt is enabled (e.g., the 18 cases we identified were fixed in the xnu-8019). Meanwhile, Apple attempts to disable interrupts before signing sensitive data. However, these fixes cannot completely mitigate this attack surface.

3) Shrinking signing interfaces (for AS❸). While protecting more sensitive data, Apple is also removing unnecessary signing interfaces. The variety of signing interfaces leads to more signing gadgets and increases the attack surface.

6.5.2 Mitigation Suggestions

1) Comprehensive sensitive data identification and binary validation (for AS❶). For unprotected sensitive data, we suggest that the XNU kernel compiler adopt a comprehensive static analysis to identify and protect all sensitive data. Besides, the compiled binary should also be validated to prove that the compilation does not introduce any security problem (e.g., sensitive data being spilled into the stack).

2) Interrupt context protection (for AS❷). To mitigate security issues caused by interrupts, we suggest signing the entire interrupt context or improving the compiler to avoid sensitive data being propagated into interrupt-unsafe registers when the interrupt is enabled.

3) Per-process key encryption (for AS❹). To improve process-dependent key management without hardware changes, we suggest encrypting the keys based on immutable process-specific information so that an attacker can not replace the keys of different processes at will.

7 Related Work

Reverse engineering on Apple M1. There have been hardware reverse engineering works on Apple M1 in recent years. The [37,40] reverse-engineer the micro-architectural CPU features such as data memory-dependent prefetcher (DMP) on the M1. For the features accessed through MMIO, AsahiLinux Team [11] and Stan Skowronek [39] reverse-engineered the features, such as DART(Apple-specific IOMMU). For the CPU features accessed by system registers, Sven Peter [35] has reverse-engineered some features such as SPRR and GXF, but no method is available for analyzing PA hardware.

PA analysis on Apple Silicon. Google Project Zero [44] analyzes the PA hardware on A12. However, they can not reverse engineer Apple's specific customization to PA. Other works [16, 22–24, 41, 44] related to PA analysis on Apple Silicon mainly focus on PA software. However, since the Apple PA hardware has not been analyzed, resulting in their analysis of PA software is incomplete. In our work, we do a comprehensive analysis of PA software after analyzing Apple PA hardware implementation, including signing interface analysis, key management analysis.

PA-related researches. PA is utilized to enforce CFI in Linux kernel [18,42,43] and user-space programs [31]. Moreover, [25] implements a PA-based CPI (Code Pointer Integrity). Researchers [30] also try to improve the security of PA against reuse attacks by software approach. Besides utilizing PA to protect the integrity of pointers, researchers try to utilize PA to implement other hardware and software collaborative mechanisms. For example, [21,29] implements sanitizers to catch memory safety bugs based on PA. Moreover, [34] combines PA and MTE (Memory Tagging Extension) to implement hardware-based isolation in Linux Kernel.

8 Conclusion

This paper conducts an in-depth reverse engineering study of PA implementation and usage on Apple M1. We develop a m1n1-based reverse engineering framework and propose multiple new techniques to analyze PA hardware implementation and usage. Our study reveals that Apple M1 introduces per-VM, per-key-type, per-boot diversifiers, and extra keys to defend against cross-domain attacks. We find that XNU kernel uses nine types of modifiers for pointer signing and authentication and key management based on customized PA hardware. We further conduct a security analysis of PA-based CFI/DFI in the XNU kernel to identify attack surfaces and report these security issues to Apple responsibly. Apple has fixed these issues in a security update, assigned us a new CVE, and publicly acknowledged us on the security advisory.

Acknowledgments

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62002317).

References

[1] 08TC3WBB. Story of Jailbreaking iOS 13. In *(Blackhat EUROPE)* (2020).

[2] APPLE. Pointerauthentication.rst. https://github.com/apple/llvm-project/blob/next/clang/docs/PointerAuthentication.rst, 2019.

[3] APPLE. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/1/web/1#sec0167b469d, 2021.

[4] APPLE. Preparing your app to work with pointer authentication. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication, 2022.

[5] APPLE. xnu. https://github.com/apple-oss-distributions/xnu, 2022.

[6] ARM. https://developer.arm.com/documentation/102142/0100/Virtualization-host-extensions, 2019.

[7] ARM. https://developer.arm.com/documentation/ddi0601/2021-12/AArch64-Instructions/AT-S1E1W--Address-Translate-Stage-1-EL1-Write, 2021.

[8] ARM, I. Sctlr_el1, system control register (el1). https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SCTLR-EL1--System-Control-Register--EL1-, 2021.

[9] ARM LTD. Arm architecture reference manual for a-profile architecture. (ARM DDI 0487I.a). https://developer.arm.com/documentation/ddi0487, 2020.

[10] ARM LTD. Developments in the arm a-profile architecture: Armv8.6-a. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-developments-armv8-6-a7, 2020.

[11] ASAHILINUX. Asahi linux. https://asahilinux.org/, 2020.

[12] ASAHILINUX. m1n1: an experimentation playground for apple silicon. https://github.com/AsahiLinux/m1n1, 2021.

[13] ASAHILINUX. Progress report: January / february 2021. https://asahilinux.org/2021/03/progress-report-january-february-2021, 2021.

[14] ASAHILINUX. Hw: Arm system registers. https://github.com/AsahiLinux/docs/wiki/HW:ARM-System-Registers, 2022.

[15] AVANZI, R. https://eprint.iacr.org/2016/444.pdf, 2016.

[16] AZAD., B. iOS Kernel PAC, One Year Later. In *(Blackhat USA)* (2020).

[17] CRESWELL, J. W., AND CRESWELL, J. D. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[18] DENIS-COURMONT, R., LILJESTRAND, H., CHINEA, C., AND EKBERG, J.-E. Camouflage: Hardware-assisted cfi for the arm linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), IEEE, pp. 1–6.

[19] DIVERTO. Macos two-machine kernel debugging. https://www.diverto.hr/en/blog/2022-03-06-macos-two-Machine-kernel-debugging/, 2022.

[20] FAN., Z. Everything has Changed in iOS 14, but Jailbreak is Eternal. In *(Blackhat USA)* (2021).

[21] FARKHANI, R. M., AHMADI, M., AND LU, L. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 1037–1054.

[22] GRASSI, M., AND CHEN, L. 2PAC 2Furious: Envisioning an iOS compromise in 2019. In *(infiltrate 2019)* (2019).

[23] HENZE., L. Fugu14 - untethered ios 14 jailbreak. https://github.com/LinusHenze/Fugu14, 2021.

[24] HENZE., L. Fugu15 - the journey to jailbreaking ios 15.4.1. https://objectivebythesea.org/v5/talks/OBTS_v5_lHenze.pdf, 2022.

[25] ISMAIL, M., QUACH, A., JELESNIANSKI, C., JANG, Y., AND MIN, C. Tightly seal your sensitive pointers with PACTight. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 3717–3734.

[26] KETTENIS, M. Re: [patch] armv8: Fix tcr 64-bit writes. https://www.mail-archive.com/u-boot@lists.denx.de/msg442041.html, 2022.

[27] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News 42*, 3 (jun 2014), 361–372.

[28] KRSTIC., I. Behind the Scenes of iOS and Mac Security. In *(Blackhat USA)* (2019).

[29] LI, Y., TAN, W., LV, Z., YANG, S., PAYER, M., LIU, Y., AND ZHANG, C. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication.

[30] LILJESTRAND, H., NYMAN, T., GUNN, L. J., EKBERG, J.-E., AND ASOKAN, N. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 357–374.

[31] LILJESTRAND, H., NYMAN, T., WANG, K., PEREZ, C. C., EKBERG, J.-E., AND ASOKAN, N. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 177–194.

[32] LLVM. https://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html, 2019.

[33] MACLACHLAN., D. Kernel debugging on apple silicon. https://dmaclach.medium.com/kernel-debugging-on-apple-silicon-ff5aa76c4429, 2021.

[34] MCKEE, D., GIANNARIS, Y., PEREZ, C. O., SHROBE, H., PAYER, M., OKHRAVI, H., AND BUROW, N. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS* (2022), vol. 22, pp. 1–17.

[35] PETER, S. Apple silicon hardware secrets: Sprr and guarded exception levels (gxf). https://blog.svenpeter.dev/posts/m1_sprr_gxf/, 2021.

[36] QUARKSLAB. An overview of macos kernel debugging. https://blog.quarkslab.com/an-overview-of-macos-kernel-debugging.html, 2019.

[37] RAVICHANDRAN, J., NA, W. T., LANG, J., AND YAN, M. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 685–698.

[38] RUTLAND., M. ARMv8.3 Pointer Authentication. In *(Linux Security Summit)* (2017).

[39] SKOWRONEK., S. Reverse Engineering the M1. In *(Blackhat USA)* (2021).

[40] VICARTE, J. R. S., FLANDERS, M., PACCAGNELLA, R., GARRETT-GROSSMAN, G., MORRISON, A., FLETCHER, C. W., AND KOHLBRENNER, D. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), pp. 1491–1505.

[41] WANG., T. Attacking iPhone XS Max. In *(Blackhat USA)* (2019).

[42] YANG, Y., ZHU, S., SHEN, W., ZHOU, Y., SUN, J., AND REN., K. Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels. https://arxiv.org/pdf/1912.10666.pdf.

[43] YOO, S., PARK, J., KIM, S., KIM, Y., AND KIM, T. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 89–106.

[44] ZERO, G. P. Examining pointer authentication on the iphone xs. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[45] ZERO, G. P. Ktrw: The journey to build a debuggable iphone. https://googleprojectzero.blogspot.com/2019/10/ktrw-journey-to-build-debuggable-iphone.html, 2019.

A Experiment Details

Table 6: Key transformation result. The result of each cell is got by setting VMKEY to a particular value, and then set each key register to 0. Finally, through bypassing hardware-based read protection, we can read the real key value of each key register, which has been transformed when Apple PA is enabled.

VMKEY	Transformation Result of					
	IB	IA	DB	DA	EX	GA
0b000	0x7d7b0db350f67ff6	0xfb0b271a781b4e27	0xe2ee9eaaa4ec5479	0x3e2b1b189fbc10b4	0xb455818159de0818	0x92584a68198c0286
	0xf60db0dcb07eb1b1	0xf625c898230bb934	0x3cd6dc8228c5488d	0xe97d268ae2681267	0x5809bcf5f3e87070	0xd8b34f463af4b03c
0b001	0xfb0b271a781b4e27	0x7d7b0db350f67ff6	0x3e2b1b189fbc10b4	0xe2ee9eaaa4ec5479	0x92584a68198c0286	0xb455818159de0818
	0xf625c898230bb934	0xf60db0dcb07eb1b1	0xe97d268ae2681267	0x3cd6dc8228c5488d	0xd8b34f463af4b03c	0x5809bcf5f3e87070
0b010	0xe2ee9eaaa4ec5479	0x3e2b1b189fbc10b4	0x7d7b0db350f67ff6	0xfb0b271a781b4e27	0x70e4228e70a3f8ff	0x5eaaa2f0e48ef187
	0x3cd6dc8228c5488d	0xe97d268ae2681267	0xf60db0dcb07eb1b1	0xf625c898230bb934	0x9cc19db7de935d05	0x982cdffcf13dfb43
0b011	0x3e2b1b189fbc10b4	0xe2ee9eaaa4ec5479	0xfb0b271a781b4e27	0x7d7b0db350f67ff6	0x5eaaa2f0e48ef187	0x70e4228e70a3f8ff
	0xe97d268ae2681267	0x3cd6dc8228c5488d	0xf625c898230bb934	0xf60db0dcb07eb1b1	0x982cdffcf13dfb43	0x9cc19db7de935d05
0b100	0xb455818159de0818	0x92584a68198c0286	0x70e4228e70a3f8ff	0x5eaaa2f0e48ef187	0x7d7b0db350f67ff6	0xfb0b271a781b4e27
	0x5809bcf5f3e87070	0xd8b34f463af4b03c	0x9cc19db7de935d05	0x982cdffcf13dfb43	0xf60db0dcb07eb1b1	0xf625c898230bb934
0b101	0x92584a68198c0286	0xb455818159de0818	0x5eaaa2f0e48ef187	0x70e4228e70a3f8ff	0xfb0b271a781b4e27	0x7d7b0db350f67ff6
	0xd8b34f463af4b03c	0x5809bcf5f3e87070	0x982cdffcf13dfb43	0x9cc19db7de935d05	0xf625c898230bb934	0xf60db0dcb07eb1b1
0b110	0x70e4228e70a3f8ff	0x5eaaa2f0e48ef187	0xb455818159de0818	0x92584a68198c0286	0xe2ee9eaaa4ec5479	0x3e2b1b189fbc10b4
	0x9cc19db7de935d05	0x982cdffcf13dfb43	0x5809bcf5f3e87070	0xd8b34f463af4b03c	0x3cd6dc8228c5488d	0xe97d268ae2681267
0b111	0x5eaaa2f0e48ef187	0x70e4228e70a3f8ff	0x92584a68198c0286	0xb455818159de0818	0x3e2b1b189fbc10b4	0xe2ee9eaaa4ec5479
	0x982cdffcf13dfb43	0x9cc19db7de935d05	0xd8b34f463af4b03c	0x5809bcf5f3e87070	0xe97d268ae2681267	0x3cd6dc8228c5488d

B Comparison between ARM PA and Apple PA

Table 7: Comparison between ARM PA and Apple PA. We list what we find on M1 for the hardware implementation of Apple PA and summarize these findings in a Summary.

Hardware		ARM PA	Apple Customization (Our Findings)	Summary
Control Register		`SCTLR_EL1`	`SCTLR_EL1` `APCTL_EL1.bit[2][3]` (Enable/Disable PAC computation at user/kernel) `APCTL_EL1.bit[1][4]` (Enable/Disable `EXTRAKEY_EL1` at kernel/user) `APCTL_EL1.bit[0]` (One-shot switch for enabling Apple PA)	Apple PA introduces a new control register to enable the Apple PA mode and differentiate PAC computation at user/kernel
Key Register		`ARMKey_EL1`	`ARMKey_EL1/EL12/EL2` `EXTRAKEY_EL1/EL12/EL2` (XOR with the *ARMKey* before PAC computation) `VMDIV_EL2` (XOR with the hard-coded *per-key-type salt* before key transformation)	1) Apple PA introduces two new 128-bit key register to diversify the PAC computation 2) Apple PA add system register redirection support to key registers including *ARMkey* and `EXTRAKEY`
Key Access	`msr`	write	`KeyLo_EL1` (write, 3 cycles) `KeyHi_EL1` (key transformation, 34 cycles): • EL1: Trans(`per-key-type salt` ⊕ `VMKEYLO`, `VMKEYHI`, `operator of msr keyhi_el1, lowest 64 bits of key register`) •EL2: Replace `VMKEY` with per-boot diversifier	Apple PA introduces key transformation and hardware-based read protection on Key Access instructions
	`mrs`	read	Trigger an exception	
PAC/AUT	`addPAC/Auth`	Check `SCTLR_EL1`	Check `SCTLR_EL1`, `APCTL_EL1`, `CurrentEL` `EXTRAKEY_EL1` ⊕ `ARMKey` (if needed)	Apple PA adds checks and XOR operations in pac/aut instruction
	`computePAC`	QARMA	Apple customized algorithm ((KeyLo ⊕ modifier) as one of the inputs)	