SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon

Hyerean Jang
Korea University
Seoul, Republic of Korea
hr_jang@korea.ac.kr

Taehun Kim
Korea University
Seoul, Republic of Korea
taehunk@korea.ac.kr

Youngjoo Shin
Korea University
Seoul, Republic of Korea
syoungjoo@korea.ac.kr

Abstract

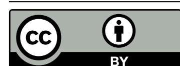
Apple silicon is the proprietary ARM-based processor that powers the mainstream of Apple devices. The move to this proprietary architecture presents unique challenges in addressing security issues, requiring huge research efforts into the security of Apple silicon-based systems. In this paper, we study the security of KASLR, the randomization-based kernel hardening technique, on the state-of-the-art macOS system equipped with Apple silicon processors. Because KASLR has been subject to many microarchitectural side-channel attacks, the latest operating systems, including macOS, use kernel isolation, which separates the kernel page table from the userspace table. Kernel isolation in macOS provides a barrier to KASLR break attacks. To overcome this, we exploit speculative execution in system calls. By using Spectre-type gadgets in system calls, an unprivileged attacker can cause translations of the attacker's chosen kernel addresses, causing the TLB to change according to the validity of the address. This allows the construction of an attack primitive that breaks KASLR bypassing kernel isolation. Since the TLB is used as a side-channel source, we reverse-engineer the hidden internals of the TLB on various M-series processors using a hardware performance monitoring unit. Based on our attack primitive, we implement SysBumps, the first KASLR break attack on macOS for Apple silicon. Throughout evaluation, we show that SysBumps can effectively break KASLR across different M-series processors and macOS versions. We also discuss possible mitigations against the proposed attack.

CCS Concepts

• **Security and privacy → Systems security**; **Operating systems security**; **Hardware reverse engineering**.

Keywords

KASLR breaking, Microarchitectural side-channel attack, Spectre-type attack


This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690189

ACM Reference Format:
Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690189

1 Introduction

Apple recently began a transition from Intel-based processors to Apple silicon, its custom-designed, proprietary ARM-based processors for its products. While the move to this ARM-based architecture increases the performance and efficiency, the inherent nature of the proprietary processor creates challenges in addressing security issues within the products. However, despite its importance, there are only a few studies on the security of Apple silicon products [32, 49, 61] compared to studies on other commodity processors [23, 25, 31, 34, 40], requiring huge research efforts into the security of Apple silicon-based systems.

In line with this, this paper studies the security of the KASLR[1] implementation on the latest Apple silicon-based macOS system. KASLR is a primary kernel hardening technique to mitigate memory corruption vulnerabilities in the kernel by randomizing the layout of the kernel address space [52]. Since its introduction, KASLR implementations have been subject to microarchitectural side-channel attacks [2, 10, 11, 23, 28, 35, 39, 40, 42, 63]. That is, using side-channel techniques on caching hardware such as TLB[2], unprivileged attackers can construct a distinguishing oracle $\mathcal{D}(v)$ that tells whether a given target kernel address v is valid (i.e., physically mapped to memory) or not. By using the oracle, the attacker finds the first valid kernel address, which determines the kernel base, and thus breaks KASLR.

To mitigate such side-channel attacks against KASLR, the latest operating systems use a kernel isolation mechanism inside the kernel [21, 22, 48]. The kernel isolation separates the kernel page table from the userspace page table. This isolation ensures that the kernel's portions of the address space are completely hidden from user-mode processes, thus thwarting any attempts to access or derive information about the kernel space. The macOS also implements the kernel isolation, denoted double map, thus provides a barrier to the previous attacks.

In this paper, we introduce *SysBumps*, the first KASLR break attack on the macOS for Apple silicon with the state-of-the-art kernel isolation technique enabled. The main idea to overcome the kernel isolation barrier is to exploit speculative execution in system calls,

[1] Kernel Address Space Layout Randomization
[2] Translation Lookaside Buffer

through which unprivileged users from the userspace can make arbitrary kernel memory access. System calls serve as interfaces to the operating system for various system services. The system call invoked in the user mode is handled by the operating system, which means that the user-provided input data is manipulated in the kernel mode. To protect the kernel from undesired user input, the system call typically performs boundary checks on arguments, especially pointer-type arguments. Hence, invoking a system call with a *kernel addresses*-given argument that is supposed to accept only userspace addresses will fail the boundary check.

We discover that during the boundary check, certain system calls in macOS exhibit a Spectre-type vulnerability [36] in the handling of user-supplied arguments. Using the Spectre gadget, an unprivileged attacker is able to create transient memory access to the attacker-supplied kernel addresses, even where the kernel address space is separated from the userspace by the kernel isolation. The transient access to the kernel address will eventually cause the TLB to change according to the validity of the address. That is, the TLB will cache the translation result for the kernel address if it is a valid address (i.e., physically mapped); otherwise, it will not. This allows the attacker to infer the validity of any chosen target kernel address v, enabling the construction of the distinguishing oracle $\mathcal{D}(v)$, our attack primitive for SysBumps attack.

Implementing the attack is not straightforward due to the lack of publicly available information about the underlying hardware as well as software. First of all, we need to understand the internals of the TLB on Apple silicon, since our attack utilizes the TLB as a side-channel source. To dissect the internals, we reverse-engineer the TLB on various M-series Apple silicon processors. Specifically, we utilize a performance monitoring unit (PMU) built into the processor for the reverse-engineering. As a result, we successfully uncover the hidden details of the TLB architecture including its set-associativity, hierarchy level, dTLB/iTLB structure, and in particular the property of dTLB that it is shared between user and kernel mode at all levels, which is sufficient to implement the prime+probe side-channel technique [59] to observe the TLB status.

To facilitate a successful attack, we also perform an in-depth analysis on the KASLR implementation in the macOS. In particular, we conduct a static code analysis of the XNU, an open-source implementation of the macOS kernel, as well as an empirical analysis on the randomness of the KASLR. Our analysis reveals that the macOS KASLR exhibits about 15 bits entropy of its randomness, which is larger than other commodity operating systems such as Linux [21] and Windows [31].

Based on our analysis results, we build an attack primitive that distinguishes between a valid and invalid kernel address. Built upon the attack primitive, we implement the SysBumps attack on the macOS for Apple silicon. Our evaluation shows that SysBumps successfully breaks the KASLR within 3 seconds with an average 96.28% accuracy under our test environments, including various M-series processors and macOS releases. The evaluation result validates the effectiveness of our attack in real-world attack scenarios, as well as its impact of KASLR break bypassing the state-of-the-art kernel isolation. We also discuss comprehensive mitigation strategies including both software and hardware solutions to counter the proposed attack.

The source code for the SysBumps attack is publicly available at https://github.com/koreacsl/SysBumps.

Contributions of this paper. Our contributions are outlined as follows:

- We discover a Spectre-type vulnerability in system calls in macOS for Apple silicon that allows for transient access to kernel addresses through speculative execution despite kernel isolation.
- We successfully reverse-engineer the hidden internals TLB of M-series CPUs by using the PMU, revealing architectural details across various processors.
- We conduct analysis of the macOS KASLR implementation, and reveal useful information, including the entropy for the randomness, the alignment size and the range of the kernel address space.
- We present SysBumps, the first KASLR break attack on macOS for Apple silicon, and propose possible effective mitigations against our attack.

Outline. The paper is organized as follows. Section 2 provides a comprehensive background on our attack. Section 3 presents attack primitives for the KASLR break attack, including details on reverse-engineering the TLB. Section 4 presents our analysis of the macOS KASLR implementation and describes our SysBumps attack in detail. Section 5 and 6 presents potential mitigation strategies against the proposed attack, and discusses attacks on other operating systems, respectively. Section 7 presents related work. Finally, we conclude the paper in Section 8.

Responsible disclosure. We reported our attack to Apple on April 29, 2024. In response, Apple has acknowledged that they have reproduced our attack and are currently investigating its root cause.

2 Background

2.1 Address space layout randomization

Address space layout randomization (ASLR) is a security mechanism that introduces randomness into the memory address locations used by a process. This randomization impacts vital memory components such as the code, stack, and heap during process initialization. KASLR extends this concept to the kernel space, ensuring each system boot results in the non-deterministic allocation of kernel code, data, and modules.

KASLR plays a crucial role in mitigating memory corruption attacks [12, 13, 27, 65] on the kernel. These attacks typically rely on an attacker's in-depth knowledge of the kernel address space structure. By randomizing kernel addresses, KASLR makes it difficult to predict the target address, effectively mitigating these threats. The effectiveness of KASLR is measured by its entropy (i.e., the amount of randomness); the higher the entropy, the less likely it is that an attacker will be able to pinpoint the exact location needed for an exploit. The entropy of the KASLR is usually determined by the size of the kernel address space and the alignment size of the kernel pages. For example, in the x86-64 Linux kernel, the address range of the text section spans 1GiB, with the base address aligned to a 2 MiB boundary. This alignment results in a total of 512 offsets (i.e., slots), 9 bits entropy.

2.2 KASLR attack and kernel isolation

KASLR side-channel attack. Breaking KASLR is an attack that allows unprivileged attackers to obtain secret information about the kernel's memory layout, which is randomized by KASLR. In particular, attackers attempt to leak the kernel's base address, from which all sufficient information about the memory layout can be inferred. The attack utilizes a distinguishing oracle $\mathcal{D}(v)$ on a kernel address v. Given a target kernel address v, the oracle returns information about v whether it is a valid address or not. To infer the kernel base address, the attacker checks whether $\mathcal{D}(v) = \texttt{valid}$ for each $v \in \mathcal{K}$, where $\mathcal{K}$ is the kernel address space. The offset of the first observed valid kernel address in $\mathcal{K}$ determines the kernel base address.

Due to memory protection, it is a challenge for unprivileged attackers to construct the oracle, as direct access to kernel memory is prohibited. To overcome this, recent studies utilize microarchitectural side-channel analysis techniques [24, 36, 39, 43, 44, 47, 51, 58, 59, 69] on constructing the oracle. In particular, they exploit the fact that only translations for valid kernel addresses are loaded onto the TLB. After attempting to indirectly load memory at v from a user process, they inspect the cache status by side-channel analysis on the TLB; only a valid address would cause the TLB status to change. Previous work has exploited design or implementation flaws in processors to indirectly induce kernel memory access from a user process. For example, certain works [23, 40] use unprivileged prefetch instructions in x86 processors to indirectly gain access to kernel memory.

Kernel page table isolation. The previous implementation of operating systems is vulnerable to the microarchitectural side-channel attacks on KASLR. The main reason for this is that these operating systems maintain both kernel and user address translation in a single page table [18, 21]. Such a design cannot prevent indirect access to kernel memory by a user process using means such as unprivileged prefetch instructions. The fundamental solution to mitigate this vulnerability is to change its design to split a page table into two tables, one for the kernel and one for the user address space, thus isolating the kernel from user processes [22]. Using the same design principle, modern operating systems have their own implementations of kernel isolation; for instance, there is KPTI [22] for Linux, Kernel Virtual Address (KVA) Shadow [48] for Windows and double map [21] for macOS.

In our work, we focus on double map for macOS with Apple silicon. This implementation leverages Translation Table Base Registers (TTBRs), which are integrated within the ARMv8-64 architecture, to achieve the desired kernel isolation. The TTBRs are a set of registers responsible for storing the base addresses of the address translation tables. They usually consist of two registers: TTBR0 for user process page tables and TTBR1 for the kernel. The macOS for Apple silicon utilizes these TTBR registers to separate address spaces, permitting access depending on the mode. In essence, kernel addresses can exclusively be accessed through TTBR1 when operating in kernel mode.

The double map implementation of the ARM-based macOS effectively prevents translation for kernel addresses from unprivileged attackers. Consequently, any such attempted access does not result in the caching of the target address in TLB, eliminating the possibility of exploiting timing differences based on kernel address validity. This strategic separation of address spaces, enhanced by the double mapping method on Apple silicon, provides a robust defense against security threats.

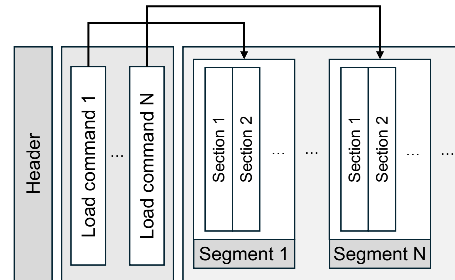


Figure 1: Mach-O binary format.

2.3 Speculative execution attacks

Modern processors incorporate speculative execution techniques to prevent pipeline stalls. This process involves predicting the next instruction to be executed based on past execution history, allowing potentially next instructions to be executed ahead of time. However, this optimization technique has serious design flaws. Incorrect predictions result in the speculative instructions and their architectural implications being discarded. However, traces of these actions are retained in the microarchitectural state, such as in the cache [36] and TLB [49], posing security risks.

Spectre attack is a type of microarchitectural side-channel attack that exploits these design flaws [7–9, 19, 25, 36, 37, 41, 64]. This attack induces incorrect predictions, resulting in the loading of security-sensitive data into the processor's cache. Attackers facilitate these mispredictions by manipulating branch predictors, which are specialized hardware designed to store the results of successful predictions for the purpose of branch prediction. Once sensitive data is maliciously loaded into the cache through this manipulated speculative execution, attackers can extract this data using cache side-channel techniques such as flush+reload [69] and prime+probe [59].

Spectre attacks are classified based on the types of branch predictors they exploit. Common variants of Spectre include Spectre-v1 [36], Spectre-v2 [7], and SpectreRSB [37, 46]. Spectre is considered much more difficult to mitigate than other microarchitectural side-channel attacks, as its root origin lies in speculative execution, a fundamental technique of modern CPUs.

2.4 Mach object file format

The Mach-O [15], short for Mach Object file format, is the standard file format for executables, object code, shared libraries, and dynamically loaded libraries within the XNU operating system framework for macOS and iOS. Figure 1 shows the Mach-O binary format, which consists of three main regions: header, load commands, and segments. The file starts with a header structure that identifies the file as a Mach-O format. The header contains information about the target architecture, file type, and necessary flags for managing instructions and data. The header is followed by the Load command, which provides the necessary metadata about the file structure and

Table 1: Devices used in our experiments.

CPU	Device	OS
M1	Mac Mini (2021)	macOS Sonoma 14.3
M1 Pro	MacBook Pro 16	macOS Ventura 13.5
M2	Mac Mini (2023)	macOS Ventura 13.2
M2 Pro	Mac Mini (2023)	macOS Ventura 13.2
M2 Max	MacBook Pro 14	macOS Ventura 13.4

system resources. These commands specify the initial layout of files in virtual memory, their associations with shared libraries, and other runtime essentials. Segments define large blocks of the file that are mapped directly into memory during execution. These include the `TEXT` segment for executable code, the `DATA` segment for writable data, and the `LINKEDIT` segment, which contains raw data used by the dynamic linker, such as symbols and strings. Each segment contains one or more sections, which are subdivisions specified to hold certain types of data.

3 Building attack primitive

To break KASLR, it is necessary to construct an attack primitive $\mathcal{D}(v)$ that determines whether a given target kernel address v is valid or not. In this section, we present the building blocks for our attack primitive targeting KASLR on macOS for Apple silicon. Our basic idea is to use TLB as a side-channel source, similar to previous side-channel attacks [11, 23, 39, 40, 42]. However, unlike the previous work, the challenge is to bypass double map, a robust kernel isolation mechanism implemented in our target system.

Our novel approach to overcome this challenge is to exploit speculative execution in system calls. Specifically, certain system calls in macOS exhibit a Spectre-type vulnerability in the handling of user-supplied arguments. Using the Spectre gadget within these system calls, an unprivileged attacker can cause translations of the attacker-supplied kernel addresses, which in turn causes the TLB to change according to the validity of the address. This allows the attacker to infer the validity of any chosen target kernel address, enabling the construction of the attack primitive that bypasses the double map.

To observe changes in the TLB state, we use the prime+probe [59] technique on the TLB. This technique involves two stages: a *prime* stage that fills the TLB with an eviction set corresponding to the target address, and a *probe* stage that probes the target set to measure the latency to determine whether the translation of the target address is cached in the TLB or not. Only the translation for a valid address is loaded into the TLB, causing an eviction of the eviction set filled at the *prime* stage. This will eventually result in measuring high latency in the *probe* stage. Conversely, any translation for an invalid address will not be cached, resulting in the low latency at the *probe* stage.

Performing a prime+probe attack requires the creation of an eviction set corresponding to the target address in the TLB. This requires an understanding of the mapping function from virtual addresses to TLB sets, as well as the structure of sets and ways in the TLB. Since the microarchitectural details of the TLB on M-series processors are not publicly known, we must reverse-engineer to obtain the necessary information.

Table 2: Event information used in TLB reverse engineering.

Event	Event number	Description
`L1I_TLB_MISS_DEMAND`	0xD4	Instruction fetches that missed in the L1 Instruction TLB
`L1D_TLB_MISS_NONSPEC`	0xC1	Load and store accesses that missed the L1 Data TLB
`L2_TLB_MISS_INSTRUCTION`	0xA	Instruction fetches that missed in the L2 TLB
`L2_TLB_MISS_DATA`	0xB	Loads and stores that missed in the L2 TLB

In Section 3.1, we provide details on our reverse engineering efforts to uncover the hidden TLB structure. In Section 3.2, we describe our exploitation technique that makes kernel address translation bypassing the kernel isolation. Finally, in Section 3.3, we implement an attack primitive based on the results described in the previous sections.

3.1 Reverse engineering TLB on Apple silicon

To use the prime+probe technique on Apple silicon's TLB, it is essential to understand several architectural properties of the TLB such as set associativity and hierarchy levels. However, necessary information about the TLB structure has not been publicly disclosed. Therefore, we perform reverse engineering to uncover the hidden properties of M-series CPUs. For this purpose, we utilize a hardware performance monitoring unit (PMU) built into the processor, to analyze the microarchitectural details of the TLB.

We implemented a performance monitoring tool based on `kperf` [68], an open source tool that provides an interface to read performance monitoring events from the M1 CPU. The PMU in Apple silicon provides about 60 performance events related to the CPU core, which can be retrieved in */usr/share/kpep/<CPU name>.plist*. For instance, performance events for M1 and M2 series chips are listed in '*a14.plist*'. Among the available performance events, we used four events related to the TLB, which are listed in Table 2, to disclose the undocumented properties.

We used various M-series processors for the reverse engineering and other experiments in the remaining sections. The devices and processors used for our work are listed in Table 1.

Set associativity. We hypothesize that the TLB is W-way S-set associative cache, where W denotes the number of ways and S denotes the number of sets. We also assume that the TLB utilizes a linearly-mapping hash function to determine a TLB set. This configuration indicates that a TLB set is identified by some consecutive lower bits of the virtual page number (VPN), by $TLB_SET(va) = VPN_{va} \, mod S$. In order to verify this hypothesis, we use the methodology proposed by Gras et al. [20].

For this, we perform experiments with several combinations of S and W. We first prepare a large size of memory block, wherein pages are sequentially allocated. Then, we perform memory accesses at W+1 pages with a stride of $S \times P$, where P is the size of a page (P is 16 KiB for macOS). While repeating memory accesses with such pattern 100,000 times, we measure `L1D_TLB_MISS_NONSPEC` event. Algorithm 1 shows the pseudo code that we used for reverse engineering the L1 dTLB structure.

Algorithm 1: Measuring performance monitoring event for TLB reverse engineering.

```
   Input: a base address of memory block base_addr, the
          number of ways W and the number of sets S
   Output: the number of measured events result
 1 function profile(base_addr, w, s):
 2     for i ← 1 to 100 000 do
 3         for j ← 1 to w + 1 do
 4             load(base_addr + w × s × page_size)
 5         end
 6     end
 7 end
 8
 9 for w ← 0 to W do
10     for s ← 0 to S do
11         start_cnt ← get_counter()
12         profile(base_addr, w, s)
13         end_cnt ← get_counter()
14     end
15     result[w][s] ← end_cnt - start_cnt
16 end
```

Figure 2 illustrates an experimental result on the M1's L1 dTLB. In the heatmap, each cell represents the number of measured events based on the combination of W and S, where the brighter color indicates a higher number of events than others. It is noteworthy that the smallest combination of W and S that triggers L1 dTLB misses is 5 and 32. The result indicates that the L1 dTLB of the M1 Pro is a 5-way 32-set associative cache with a linearly-mapping hash function.

We also analyze the set associativity of the L1 iTLB. Unlike dTLB, we allocate a large region of executable memory and then fill it with `ret` instructions. Then, we execute these instructions at the W+1 pages with a stride of $S \times P$. This results in the address translation for the code page being loaded into the L1 iTLB. The rest of the procedure is the same as for analyzing the L1 dTLB, except that this time the `L1I_TLB_MISS_DEMAND` event is measured.

Following the approach for the L1 TLB, we also analyze the set associativity of the L2 TLB. In this case, we use `L2_TLB_MISS_DATA` and `L2_TLB_MISS_INSTRUCTION` events for L2 dTLB and L2 iTLB, respectively. From the experiments, we obtain a heatmap that shows a similar pattern in Figure 2, confirming that all the tested processors use the L2 TLB of a set-associative cache with a linearly-mapping hash function.

TLB hierarchy. To verify that L2 TLB is the last level cache in Apple silicon, we conduct an experiment that investigates whether an L2 TLB miss triggers a page table walk. For the experiment, we construct an eviction set E, where $|E|=W+1$, that maps to the same L2 TLB set. Then we make a memory access to each element in E, measuring page table walk events, i.e., `MMU_TABLE_WALK_DATA` (number: 0x8), and `MMU_TABLE_WALK_INSTRUCTION` (number: 0x7). From the experimental results, we observe that the last memory access results in a page table walk, indicating that L2 TLB is the last level translation cache.

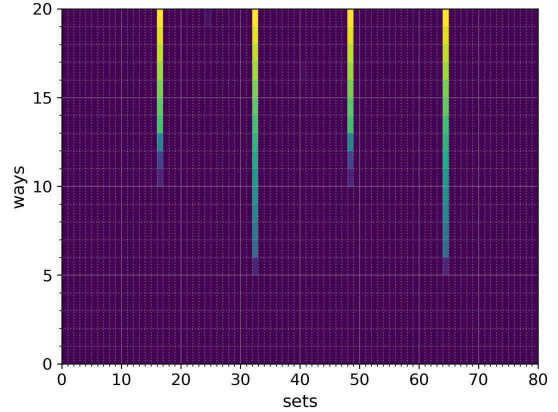


Figure 2: Heatmap graph showing the measured performance monitoring events with various combinations of W and S.

Structure of dTLB and iTLB. Now, we examine whether the TLB has a unified structure for dTLB and iTLB. For this, we perform an experiment to check whether a TLB entry allocated by code pages can be evicted by a data page, and vice versa. We construct eviction set E_{L1i} and page P_{L1d} using the same set index bits for L1 iTLB and L1 dTLB, respectively. Then, for each element of E_{L1i}, we execute a branch instruction to fill the L1 iTLB set. After that, we access P_{L1d} and, while refilling the L1 iTLB with E_{L1i}, we measure the `L1I_TLB_MISS_DEMAND` event. If the TLB has a unified structure, an increase in the event would be observed. However, our experimental result reveals no such increase in the event, suggesting a separation of L1 dTLB and L1 iTLB. Extending our investigation on L2 TLB, we similarly observe no L2 TLB miss events (i.e., `L2_TLB_MISS_INSTRUCTION`, `L2_TLB_MISS_DATA`). Across all tested Apple silicon, our findings disclose a separation between the dTLB and the iTLB.

TLB sharing between user and kernel. Although the TLB is shared between the kernel and user processes on other processors such as Intel and AMD, it has not been thoroughly analyzed on M-series processors yet. Hence, we design an experiment to uncover this property. In the experiment, we examine whether TLB entries allocated by a user process can be evicted by a kernel process. To make a memory access in kernel space at will, we implemented a kext, a kernel extension for macOS, which enables running our executable code with kernel privileges.

First, we examine the property of dTLB. In the first step of our experiment, we choose a target address T from valid kernel addresses; this can easily be done by picking the address of a variable in our kext module. We then construct an eviction set E_{L1d} against T, consisting of userspace addresses, and fill the L1 dTLB set with E_{L1d}. Next, we perform a memory access on T through our kext module, and measure `L1D_TLB_MISS_NONSPEC` event after reloading E_{L1d}. If the L1 dTLB is shared between a user and kernel, the reload would result in L1 dTLB misses. From the experiment, we observed a noticeable increase in dTLB miss events, indicating that dTLB is shared between a user and the kernel. We also performed

Table 3: Properties of TLB on Apple silicon.

	M1	M1 Pro	M2	M2 Pro	M2 Max
L1 dTLB					
Number of sets	32	32	64	64	64
Number of ways	5	5	4	4	4
Hash function	Linear	Linear	Linear	Linear	Linear
Sharing with kernel space	✓	✓	✓	✓	✓
L1 iTLB†					
Number of sets	32	32	32	32	32
Number of ways	6	6	6	6	6
Hash function	Linear	Linear	Linear	Linear	Linear
Sharing with kernel space	✗	✗	✗	✗	✗
L2 dTLB					
Number of sets	256	256	256	256	256
Number of ways	12	12	12	12	12
Hash function	Linear	Linear	Linear	Linear	Linear
Sharing with kernel space	✓	✓	✓	✓	✓
L2 iTLB†					
Number of sets	256	256	256	256	256
Number of ways	12	12	12	12	12
Hash function	Linear	Linear	Linear	Linear	Linear
Sharing with kernel space	✗	✗	✗	✗	✗

† iTLB for userspace

the experiment for L2 dTLB, and obtained the same result as for the L1 dTLB.

We next examine the property of the iTLB. For this, we implement a function `foo()` in our kext module and set the target address T to the start address of the function. Then, we construct an eviction set E_{L1i} against T, consisting of branch instructions in userspace, and fill the L1 iTLB set by jumping to each element in E_{L1i}. After that, our kext module performs branching to T and measures `L1I_TLB_MISS_DEMAND` event while branching to each element in E_{L1i} again. In this case, we cannot observe any TLB miss events on L1 iTLB from the experiment. The same result was obtained for the L2 iTLB. This observation indicates that the TLB has separate iTLBs for a user and the kernel.

To sum up, Table 3 lists the disclosed TLB properties of Apple silicon, and Figure 3 illustrates the TLB structure. It is noteworthy that a user and kernel share the same dTLB, potentially resulting in contention on it. With this knowledge, we observe the kernel's memory access behavior through dTLB to break KASLR.

Comparison to other work. Ravichandran et al. [49] attempted to reverse engineer the internals of the TLB on Apple silicon, which is similar to our work, but has a different research goal of exposing the ARM pointer authentication vulnerability. The main difference in the reverse-engineering approach is that while they rely on a timestamp counter, we use a PMU for our analysis. Because the PMU provides rich data on microarchitectural behavior, we were able to obtain more detailed information on the TLB, which even gave us slightly different results from the previous work. For instance, the authors claim that a TLB entry evicted from L1 iTLB is loaded into L1 dTLB due to their hierarchical relationship [49]. To verify this, we conducted an experiment with our M1 CPU-equipped device. First, we constructed an eviction set E for the L1 iTLB, where $|E| = W + 1$, and W is the number of ways of the cache. For the M1 CPU, $|E|$ is 7 because W for the L1 iTLB is 6. Next, we filled the iTLB with

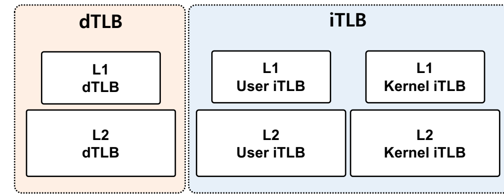


Figure 3: Hierarchy of TLB on Apple silicon.

E by executing a branch for each element $e \in E$. This triggers an eviction from the L1 iTLB set. During the experiment, we measured the counter `L1D_TLB_FILL` (number: 0x5). If the L1 dTLB serves as a backing cache for the L1 iTLB, an increase in the counter might be observed because the last element in E results in an entry allocation in the L1 dTLB. However, we have not observed such results.

Recently, Apple released the Apple Silicon CPU Optimization Guide [16], concurrent with our work[3]. The document contains information on the TLB structure of various M-series processors. We confirm that our findings about the TLB, such as the TLB hierarchy (e.g., L1 and L2 TLBs) and the TLB size (e.g., 3,072 entries in the L2 TLB), are consistent with the document. However, the document lacks more detailed information about the TLB internals, such as set associativity, mapping function, and user/kernel separation, which are necessary to implement the prime+probe attack. In our work, we performed reverse engineering and uncovered those hidden details about the TLB. For instance, we revealed that the L2 dTLBs in all M-series processors commonly have 256 sets and 12 ways on each, with a linear mapping function, shared between user and kernel mode. Those information is not contained in the Apple's document. Our findings from the reverse engineering efforts provide the essentials for implementing our attack.

3.2 Speculative execution inside system call

The kernel isolation in macOS prevents unprivileged users from causing translation for arbitrary kernel addresses, which is essential for the construction of an attack primitive. Our approach to bypassing kernel isolation is to use certain system calls that take arguments from user-supplied addresses to indirectly cause access to a kernel address. However, these system calls internally perform validation checks on the user input, such as boundary checking or address range checking, to filter out any undesired input [55, 66, 67]. Because of the validation checks, invoking the system calls with invalid arguments, such as specifying a kernel address for arguments that only accept userspace addresses, will not result in kernel memory access.

However, we find that it is possible to bypass this validation by inducing misspeculative execution within the system call. This approach is similar to the Spectre-v1 [36] technique, which transiently manipulates the control flow of conditional branches. To illustrate this, we give an example of a `chdir(user_addr_t path)` system call. It takes a string address `path` as an argument and passes it to the `copyinstr()` function. Figure 4 shows a snippet of code from `copyinstr()`, where `user_addr` is the user input address provided by `chdir()`. The `copy_validate()` function actually performs a

[3]The publication date of the document is March 21, 2024.

Figure 4: Code snippet that validates user input address.

```
int copyinstr(const user_addr_t user_addr, char *kernel_addr, vm_size_t nbytes, vm_size_t *lencopied)
{
    int result;
    ...
    result = copy_validate(user_addr, (uintptr_t)kernel_addr, nbytes, COPYIO_IN);
    if (__improbable(result)) {
        // When user_addr is invalid
        return result;
    }

    // When user_addr is valid
    user_access_enable();
    result = _bcopyinstr((const char *)user_addr, kernel_addr, nbytes, &bytes_copied);
    user_access_disable();
    ...
}
```

validation check on `user_addr` (at line 5 in Figure 4) to see if the address falls within the user's address space. The conditional branch in line 6 determines subsequent control flows based on the results of the bounds check. If the `user_addr` is valid, it is passed to the `_bcopyinstr()` function (line 13), which copies the data stored in `user_addr` to the kernel memory space (`kernel_addr`). Otherwise, the function returns immediately (line 8).

We find a Spectre-type gadget inside the `copyinstr()` function around the conditional branch in line 6. With the gadget, we create a speculative execution on the system call to make a transient memory access to a target kernel address. Specifically, through the deliberate mistraining of the branch predictor, we can trigger transient execution of the memory access, regardless of the validation result. Although this execution will be subsequently rolled back, it leaves a trace in the TLB if the target address is valid (i.e., mapped to physical memory). Consequently, these traces allow us to determine whether a target kernel address is valid or not.

Validation of Spectre-type gadgets. To validate that transient access to kernel addresses really does occur in the Spectre-type gadgets in Figure 4, we perform an additional experiment. To do this, we modified the macOS kernel binary to insert certain barrier instructions `DSB` and `ISB` for ARM64 before the conditional branch (between lines 5 and 6 in Figure 4). The barrier instructions prohibit any speculative execution in the conditional branch. We then compare the experimental result of our attack primitive on the unmodified macOS, which is detailed in the next section, with the result on the patched macOS. From the comparison, we see no timing difference on the patched macOS, while we see a timing difference (Figure 6) on the unmodified version, indicating that transient memory access occurs in the conditional branch.

3.3 Attack primitive

Based on the observation of speculative execution in system calls, we can build an attack primitive that allows us to know whether a given target kernel address v is valid or not. This attack primitive uses the prime+probe technique on the TLB. To implement the prime+probe attack, we leverage the knowledge gained from reverse-engineering the TLB described in Section 3.1. Specifically, we construct an eviction set that targets the L1 or L2 dTLB as it is shared between user and kernel space, allowing us to evict kernel addresses using the eviction set of user addresses. Figure 5 shows a code snippet for our attack primitive. It consists of four steps described below.

Figure 5: Pseudocode of attack primitive.

```
/*-----------------------------------------------------------
 char * kernel_addr : a target kernel address (v)
 int sys_num        : a system call number
-----------------------------------------------------------*/
int validity_test (char * kernel_addr, int sys_num){
    // Step1. training with an arbitrary userspace address
    syscall(sys_num, user_addr);
    syscall(sys_num, user_addr);

    // Step2. priming the TLB with an eviction set
    prime();

    // Step3. invoking a system call with v
    syscall(sys_num, kernel_addr);

    // Step4. probing the TLB state
    if( probe(kernel_addr) < THRESHOLD )
        return false; // v is an invalid address.
    else
        return true; // v is a valid address.
}
```

Step1 (Training) Invoke system calls with an argument of a valid user address (lines 7-8), to mistrain the conditional branch within the system call. For Apple silicon processors, two invocations of system calls are enough to train the branch predictor.

Step2 (Prime) Fill the dTLB set corresponding to v with the eviction set (line 11).

Step3 (Access) Invoke the system call again, but with an argument of v (line 14). This will lead to a misprediction in the internal conditional branch, subsequently resulting in a transient execution that performs memory access to v. The execution may affect the dTLB depending on the target address; the dTLB will load a translation of v if it is valid, otherwise, not.

Step4 (Probe) Probe the eviction set and measure its latency (line 17). The measured latency reflects the state of the dTLB; A cycle higher than the predetermined threshold indicates that v is physically backed up (i.e., a valid address), while a cycle lower than the threshold indicates that v is not.

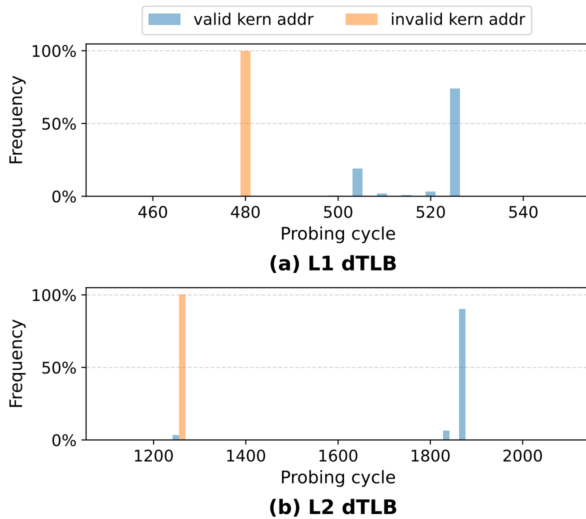


Figure 6: Measurement for two kernel addresses on the M1 CPU.

The threshold used in Step 4 varies depending on the TLB levels. In the attack, the threshold can be determined through multiple tests.

Evaluation. To validate its effectiveness, we performed an experiment for our attack primitive with two kernel addresses: v_1, which is physically backed (i.e., a valid address), and v_2, which is not (i.e., an invalid address). Figure 6 (a) and (b) show the probing cycles measured for these addresses on the L1 and L2 dTLB on the M1 CPU, respectively. For a valid address v_1, denoted '`valid kern addr`' in the figure, it shows high probing cycles as the address has evicted one of our elements primed at Step2, resulting in high latency in probing at Step4. On the other hand, it has low probing cycles for the invalid address v_2, denoted '`invalid kern addr`', as its translation has not been cached on the TLB. We obtained the same experimental results for all the devices listed in Table 1.

We also observe from the result that probing cycles for v_1 and v_2 are more distinguishable on L2 dTLB than L1 dTLB. We attribute this to the different inherent properties of these TLBs. For L1 dTLB, its small size leads to frequent evictions, resulting in a high chance of false positives. In addition, the difference in L1 dTLB probing cycles based on address validity is less than that of L2 dTLB, which could lead to high error rates in TLB probing due to noise. Indeed, our tests have shown that attacks monitored at the L1 dTLB level significantly drop in accuracy compared to those conducted at the L2 dTLB level. In conclusion, we decide to use the L2 dTLB in our attack primitive to achieve better accuracy in SysBumps attack introduced in Section 4.

Identifying exploitable system calls. We manually investigate to identify vulnerable system calls that can be exploited as our attack primitives. The '*syscall.master*' file in the XNU source code [6] enumerates all 565 system calls available for the macOS. We look for system calls that take pointer-type arguments, excluding those related to inter-process communication (IPC) and process management, as they may have unexpected side effects. This leaves 80 system calls.

To verify the exploitability of the remaining 80 system calls, we performed experiments with our attack primitive using these system calls. We succeeded in the attack with 25 out of them, such as `chdir()` and `fgetxaddr()`, all of which are listed in Appendix A. These vulnerable system calls typically take pointer-type arguments. For instance, `chdir()` takes a path to the specified directory, which is of type `const char*`. `fgetxaddr()` takes an attribute value of type `user_addr_t`, which is also one of pointer types.

To examine the underlying root cause, we analyze the XNU source code of the vulnerable system calls. Through the analysis, we discover that `copyinstr()` or `copyin()` functions are internally invoked within all these system calls. Both functions aim to copy data from userspace to kernel space. We also find out Spectre-type gadgets inside both `copyinstr()` and `copyin()`, as shown in Figure 4.

4 Breaking KASLR on macOS for Apple silicon

In this section, we first examine the implementation details of KASLR in macOS for Apple Silicon. Next, we analyze the kernel memory layout of the macOS using the attack primitive that we build in Section 3. Finally, based on these analyses, we implement SysBumps, our KASLR breaking attack targeting macOS for Apple silicon.

4.1 KASLR entropy analysis

The KASLR breaking attack requires an understanding of the KASLR implementation including the range of kernel base addresses and its alignment size. Since details of the underlying implementation in macOS have not been disclosed, we attempt to uncover it through both static code analysis and empirical analysis on the XNU kernel.

Static code analysis. First, we analyze the source code of the XNU in an attempt to identify the KASLR implementation. From the source code [4], we find that the kernel base address is determined as follows.

$$Kernel_base = \texttt{0xfffffe0007004000} + slide. \quad (1)$$

The kernel base address is actually dependent on the slide, which is an offset randomly generated at boot time. However, we have not been able to find the implementation of slide generation in the source code, suggesting that it is outside the kernel and that a bootloader is in charge of slide generation.

Empirical analysis. As the actual implementation of the random generation of slide is not made public, we decide to perform an empirical analysis on its distribution. Specifically, we try to figure out the actual range of slide by measuring the allocated kernel base addresses, from which the value of slide is determined by Eq.1. To do this, we implemented another kext module that records the current kernel base address at boot time. In the experiment, we collected 50,000 different kernel base addresses for each device listed in Table 1.

Figure 7 shows the distribution of these collected kernel base addresses, and Table 4 presents the measurement results in detail. The result shows that all tested devices have the same maximum

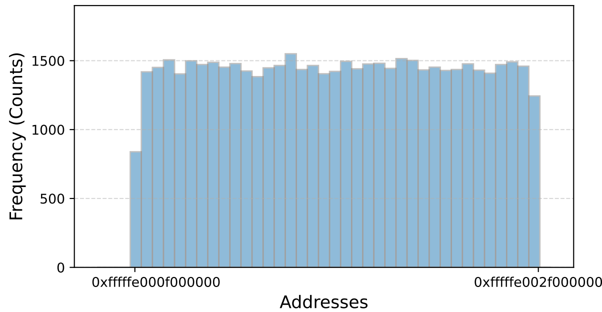


Figure 7: Distribution of kernel base addresses measured over 50,000 reboots.

Table 4: Measurements for kernel base range.

CPU	Min	Max	GCD	#Slots (bits)
M1	0xfffffe000f3f4000	0xfffffe002f000000	0x4000	32,515 (14.98)
M1 Pro	0xfffffe000f0f4000	0xfffffe002f000000	0x4000	32,707 (14.99)
M2	0xfffffe000f0e0000	0xfffffe002f000000	0x4000	32,712 (14.99)
M2 Pro	0xfffffe000f1bc000	0xfffffe002f000000	0x4000	32,657 (14.99)
M2 Max	0xfffffe000f1c4000	0xfffffe002f000000	0x4000	32,655 (14.99)

kernel base address, while they have different minimum addresses. We also observe from the result that the greatest common divisor (GCD) of these collected kernel base addresses is 16 KiB, aligning with the system's page size. It is important to note that the GCD of kernel addresses is a multiple of the alignment size, and these alignment sizes are typically set as a multiple of the page size for system performance reasons. Consequently, based on these findings, we can infer that the alignment size for kernel base addresses is indeed set to 16 KiB.

From our experimental observations, we determine that the actual range of kernel base addresses is at least 32,515 (= $2^{14.98}$) possible slots (i.e., the allocated unit of kernel base address), exposing approximately 15 bits of entropy to attackers.

4.2 Kernel layout analysis

To gain an insight into the construction of the SysBumps attack, we analyze how much information about the kernel memory layout can be obtained by our attack primitive. Specifically, we run the attack primitive for the full set of possible kernel slots and measure the latency (i.e., the probing cycle). All the measurements are performed on the same KASLR instance of a device equipped with an M1 CPU.

Figure 8 shows the measured latency for kernel slots (shown below in the figure) as well as the actual kernel memory layout (shown above), both of which are aligned with the slot numbers. In the figure, some regions exhibited persistently high probing cycles (e.g., the region of slot numbers between 16,384 and 17,918), indicating that they have valid kernel address spaces. On the other hand, certain regions have both high and low cycles for two consecutive slots (e.g., the region between 14,000 and 16,383), implying that they are invalid address spaces. We attribute the observation of high cycles in the invalid address region to noise generated by our attack primitive, since it involves multiple executions of system calls.

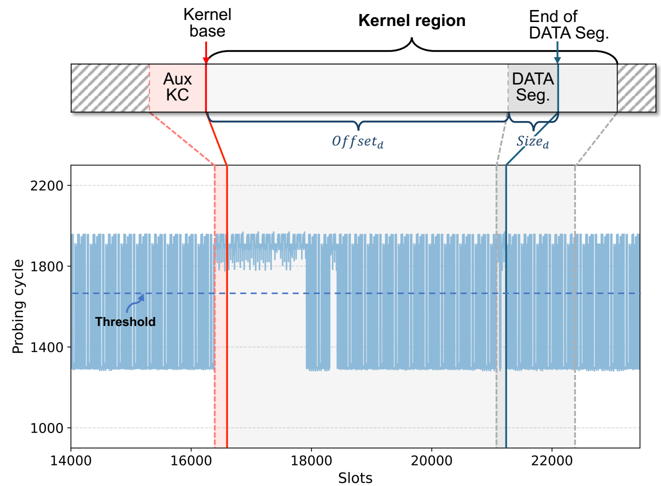


Figure 8: Probing with the attack primitive over the kernel base range.

Despite a certain amount of noise, we observe that high latency slots occur continuously in the valid address regions, allowing a clear distinction between valid and invalid areas.

As shown in Figure 8, there is a slight discrepancy between the actual kernel base, indicated by a red line in the figure, and the start slot measured as valid, indicated by a red dashed line. Further analysis reveals that the region in front of the kernel base, shown as a shaded area in red in the figure, is occupied by *Auxiliary kernel collection* (AuxKC). The AuxKC is a preserved space allocated for third-party kernel extensions (kext) [3]. The size of the AuxKC varies depending on the kext modules installed on the macOS, which is unknown to unprivileged users. The presence of such regions poses a significant challenge to finding the exact location of the kernel base, as it requires the ability to determine the size of the AuxKC region in unprivileged mode, which is quite challenging.

On the other hand, we observe that there are a number of slots that are measured as valid at the end of the kernel region (e.g., slots 21,148 - 21,233 in Figure 8). Through further analysis, we confirm that the valid slots are consistently aligned with the end of the kernel's `DATA` segment, as indicated by the blue line in the figure. The size of the `DATA` segment and its offset are always consistent throughout the main kernel image, and any unprivileged user can obtain this information by examining the load commands of a kernel cache file, which is a pre-linked executable that contains the kernel and essential drivers[4].

Attack strategy. Based on our analysis, we establish a strategy for our SysBumps attack that instead of the kernel base address, we choose to find out the end address of the `DATA` segment. Once the location of the `DATA` segment has been identified, we are able to determine the kernel base address with the information about the size and offset of the segment.

Further observation. We discover that there are certain regions (e.g., slots 17,919 - 18,303 in Figure 8) showing low cycles in the

[4]The kernel cache file is typically located at */private/var/db/KernelExtensionManagement/KernelCollections/BootKernelCollection.kc.*

Table 5: The attack performance.

	`chdir()`		`getxattr()`		`pathconf()`	
	Accuracy	**Time(s)**	**Accuracy**	**Time(s)**	**Accuracy**	**Time(s)**
M1	98.8%	2.34	96.6%	2.35	98.0%	2.33
M1 Pro	95.7%	2.54	92.1%	2.52	94.5%	2.50
M2	97.6%	2.11	97.9%	2.12	97.0%	2.09
M2 Pro	96.3%	2.21	94.9%	2.21	95.9%	2.15
M2 Max	97.0%	2.15	95.9%	2.14	96.0%	2.09

latency, thus measured as invalid, while belonging to the valid kernel regions. This suggests that the translations for kernel addresses in such regions are not cached in the TLB, which requires further analysis. However, the uncertainty about the region does not affect our KASLR breaking attack, since it is sufficient to use the end address of the DATA segment to determine the kernel base address.

4.3 SysBumps

4.3.1 Threat model. In this attack, we assume an unprivileged attacker, who can execute arbitrary code on a user process, but does not have any privileges over the operating system. The attacker aims to exploit memory corruption vulnerabilities in the kernel space using software exploitation techniques [13, 50, 56, 70], which requires bypassing KASLR. The target system employs robust kernel isolation techniques (i.e., double map) to mitigate KASLR breaking attacks using microarchitectural side-channel techniques. Therefore, the attacker's goal is to bypass KASLR by overcoming these mitigation measures.

4.3.2 Attack procedure. Following the attack strategy presented in the previous section, an unprivileged attacker performs the attack through the following steps.

Step 1. The attacker gets the information about the DATA segment of the main kernel image including its offset, denoted $Offset_d$, and the size, denoted $Size_d$. This information can be easily obtained by examining load commands in the kernel cache file with the *otool* [60], which is a tool that enumerates specific information in Mach-O format binaries.

Step 2. The attacker looks for the end address of the DATA segment of the kernel image, which is denoted $Addr_{endD}$ using the attack primitive introduced in Section 3. To find out $Addr_{endD}$, he/she starts at the address of the highest kernel slot, and continues by decreasing the slot number until the first *valid* slot, where the attack primitive (Figure 5) returns `true`, is found. Since our attack exhibits some measurement errors, the attacker repeats the examination over the kernel slots several times and then gets the averaged results to minimize the error. The attacker further reduces the false positives with a strategy that instead of looking for a single valid slot, he/she looks for continuous multiple valid slots, as this indicates high evidence of the valid DATA section.

Step 3. With the information obtained in Step 1 and 2, the attacker is able to determine the kernel base address *Kern_base* using the following equation.

$$Kern_base = Addr_{endD} - \{Offset_d + Size_d\}. \quad (2)$$

4.3.3 Evaluation. To validate the effectiveness of our attack, we evaluate its performance in terms of accuracy and execution time. The evaluation is performed with our tested devices listed in Table 1, which covers a variety of Apple silicon processors as well as macOS release versions from 13.1 to 14.3. We choose three system calls `chdir()`, `getxattr()`, and `pathconf()` for our evaluation among the possible system calls enumerated in Appendix A.

Attack accuracy. For each target device, we ran the following experiment when the device was booted; so, the kernel base remains the same under the same experiment. The experiment is that we ran the SysBumps attack 10 times, selected the most frequent answers for *Kern_base* among them, and then compared the answer with the real kernel base address. We then repeated the above experiment 100 times by rebooting the system, each time with a different kernel base address.

Table 5 shows the evaluation result. We observe from the result that an averaged success rate ranges from 94.1% to 97.8%, indicating the high accuracy in finding out the kernel base address. It is noteworthy that in the failed cases, the answers returned by the attack were only 1 or 2 slots away from the real kernel base address. We attribute this discrepancy to false positives due to the noise introduced by the attack. Even in cases of such errors, the detected kernel base was significantly close to the actual address, effectively reducing the entropy of KASLR.

Attack execution time. We also evaluate the execution time of our attack under the same experimental environments. The attack execution time means the time elapsed from the start of the attack to the time when the answer is returned. The execution time is most likely affected by the real kernel base address, since its occupied slot determines the number of slots to examine in our attack.

In light of this observation, we consider a worst-case scenario where the kernel base is located at the lowest of all possible kernel slots. Table 5 also shows the evaluation results of the execution time. From the result, we conclude that our attack is able to determine the kernel base address within about 3 seconds, showing its effectiveness in a real-world attack scenario.

5 Countermeasures

In this section, we discuss potential mitigations against the SysBumps attack at the hardware and software levels.

Partitioning dTLB between user and kernel space. Apple M-series CPUs currently employ a dTLB design that is shared by both a user and a kernel. This design poses a security concern, as an attacker can infer memory access patterns of the kernel by observing contention on the dTLB. Hence, one possible mitigation approach is to partition the dTLB between a user and a kernel, completely eliminating contention on the dTLB. As shown in Section 3.1, we revealed that the iTLB of Apple silicon is already partitioned into two separate hardware resources based on the privilege level. This prevents the SysBumps attack from using the iTLB to break KASLR [39, 40]. Similarly, we expect that employing such a design to dTLB would be effective in mitigating the SysBumps attack.

Allocating TLB entry for invalid address. The main idea behind SysBumps attack for breaking KASLR is to distinguish valid addresses from invalid ones. This has been achieved by the TLB design, where the TLB allows an unprivileged user to allocate a TLB entry for kernel space as long as the memory address is

valid. By observing the change in the TLB state, an attacker can identify the base address of the kernel. Most microarchitectural side-channel attacks that de-randomize KASLR also rely on this methodology [21, 31, 39, 40]. Therefore, modifying the TLB design to allocate TLB entries even for invalid addresses could be an effective mitigation. With this approach, SysBumps is unable to break KASLR since an attacker can no longer differentiate valid kernel addresses from invalid ones through the TLB. We believe that this mitigation could prove effective against existing attacks that exploit the TLB design to break KASLR.

Preventing speculative execution with a fence. SysBumps attack exploits speculative execution that occurs during the execution of system calls. Given that the SysBumps gadget aligns with Spectre-v1, one possible short-term solution is to insert serializing instruction, such as `DSB`, `ISB` for ARM64, before the conditional branch (e.g., line 6 in Figure 4) [30, 38, 62]. We confirmed earlier in Section 3.3 that, after applying this mitigation, the timing differences depending on the address validity are no longer detected.

While this mitigation effectively protects against threats arising from speculative execution, it may lead to some performance degradation when performing system calls. To evaluate this impact, we measured the execution time of system calls on macOS with and without employing the mitigation. We focused on three system calls—`chdir()`, `getxattr()`, and `pathconf()`—previously used for performance evaluation. We executed each system call 100,000 times across different kernel versions. The experimental results indicated that the performance overhead caused by the serializing instruction is so negligible that the difference between the pre- and post-mitigation states is indistinguishable. This result demonstrates that inserting serializing instructions can be employed without significant performance compromise. However, this solution does not address the root cause of the SysBumps attack, allowing an attacker to break KASLR with other potential code gadgets.

Restricting speculative execution window to reach target instruction. While SysBumps utilizes code gadgets similar to Spectre-v1, the control flow executed during speculative execution differs. Unlike Spectre-v1, the SysBumps attack aims to speculatively execute instructions outside of an if statement, specifically in the control flow where the conditional branch is false. Therefore, one solution is to allow speculative execution but limit the execution of the target instruction within the speculative execution window. This can be achieved by reordering the code sequence to place the target instruction farther away from the conditional branch, beyond the reach of the speculative execution window. Fortunately, as we will describe in Section 6, macOS running on Intel processors do not perform memory access with an attacker-controlled input immediately after the conditional branch. As a result, the target instruction is not within the speculative execution window, preventing SysBumps attack from breaking KASLR. Employing a similar approach to macOS running on Apple silicon can serve as a short-term defense against SysBumps attack.

6 Discussion

In this section, we investigate the feasibility of the SysBumps attack on different operating systems. To this end, we have performed this attack on macOS, Ubuntu Linux, and Windows on Intel processors.

Attack on macOS for Intel CPUs. In the experiment, we attempted the SysBumps attack on macOS Ventura 13.6 with an Intel Core i9-9800H processor. As a result, SysBumps failed to determine the base address of the macOS kernel. To understand this failure, we performed a detailed analysis of the system call implementation in macOS for an x86-64 system [5]. From the analysis, we find that there is a significant difference of the system call implementation between macOS for Apple silicon and macOS for x86-64. For the macOS x86-64, memory accesses with the system call arguments do not occur immediately after conditional branches in the validation check. As a result, these memory accesses do not happen within the speculative execution window, making the attack infeasible.

Attack on Ubuntu Linux. For this attack, we set up an experimental environment on an Intel Core i5-9600K running Ubuntu 20.04.6 LTS (kernel version: 5.15.0-91-generic). SysBumps also failed to determine the base address from userspace. Throughout the analysis, we could not find any Spectre-type gadgets in the system call implementation in the Linux kernel, unlike the macOS for Apple silicon, which has the gadgets as shown in Figure 4.

Attack on Windows. We attempted the attack on Windows 10 (version: 22H2) running on an Intel core i9-10900. The attack failed because invoking a system call with arguments of arbitrary kernel addresses results in a segmentation fault; this different behavior is not observed in other operating systems.

7 Related Work

KASLR side-channel attacks. Ongoing research efforts have been made to break KASLR through microarchitectural attack vectors. The key aspect of the attack is to create an attack primitive that distinguishes between valid and invalid kernel addresses. All of the work commonly exploit the state change in the microarchitecture such as TLB, but have different approaches in building the attack primitive. Table 6 provides a summary of these attacks.

Most previous work attempts to infer the validity of the target address by observing the TLB state, exploiting the fact that the TLB only caches valid kernel address translations. Double page fault [28] and Drk [31] attacks take advantage of the timing difference in handling page faults, where the difference comes from the way handles valid and invalid addresses. The attacks proposed by Gruss et al. [21] and Lipp et al. [40], and the EntryBleed attack [45] utilize the timing difference in the execution of unprivileged prefetch instructions on x86 processors to observe the TLB state. AVX-TSCHA [33] attack exploits AVX's masked operations which have timing differences in its execution depending on the validity of target kernel address. PLATYPUS [42] and the attack proposed by Lipp et al. [40] exploit the difference in power consumption, which is correlated with the TLB state, instead of the execution time using the Running Average Power Limit (RAPL) interface [29]. Evtyushkin et al.'s work [17] and TagBleed [39] showed de-randomizing KASLR by inducing collisions in BTB and TLB, respectively.

Some of the work is aimed at making access to kernel addresses by creating transient executions. These attacks leave a detectable trace for valid kernel addresses during the transient execution, allowing attackers to infer the validity of the address. The Flush-Conflict attack [63] exploits Intel's non-temporal moves (`MOVNT`)

Table 6: Comparison of existing KASLR breaking attacks.

Attack	CPU	OS			Entropy† (bit)	Leakage source	Technique
		Linux	Windows	macOS			
Double page fault [28]	Intel, AMD	✓	✓		9 / 13	TLB	Page fault handling
Gruss et al [23]	Intel		✓		13	TLB	Software-based prefetch
Evtyushkin et al. [17]	Intel	✓			9	Branch target buffer	Transient execution
Drk [31]	Intel	✓	✓	✓*	9 / 13 / 8	TLB	Intel TSX
Data Bounce [10]	Intel	✓			9	TLB	Transient execution
TagBleed [39]	Intel	✓			9	TLB	TLB information leak
EchoLoad [11]	Intel	✓	✓		9 / 13	Reorder buffer	Transient execution
PLATYPUS [42]	Intel	✓			9	TLB	RAPL interface
FlushConflict [63]	Intel	✓			9	TLB	Transient execution
Lipp et al. [40]	AMD	✓			9	TLB	RAPL interface
AVX-TSCHA [33]	Intel, AMD	✓	✓	✓*	9 / 13 / 8	TLB	AVX instruction
EntryBleed [45]	Intel	✓			9	TLB	Software-based prefetch
SysBumps (This work)	Apple silicon			✓	15	TLB	Transient execution

† Linux - 9 bits / Windows - 13 bits / macOS for Intel - 8 bits / macOS for Apple silicon - 15 bits
* macOS for Intel processors

instruction, which can cause transient execution for subsequent instructions based on the validity of the target address. EchoLoad [11] exploits the vulnerability in the incomplete Meltdown mitigation that accesses to valid kernel addresses are not stalled but just zeroed out. Data Bounce [10] takes advantage of the fact that store-load forwarding is performed using inaccessible kernel addresses, even if only some of the addresses match during transient execution.

Our work is different from the previous work in two aspects. First, we mainly focus on breaking KASLR on macOS for Apple silicon, which has not been explored yet in the research community. Although some previous works [31, 33] have demonstrated attacks against KASLR in macOS, they exploited hardware flaws in x86 processors, not Apple silicon. Moreover, these attacks have already been mitigated in the latest version of macOS by the kernel isolation. To the best of our knowledge, our work introduces the first attack that successfully breaks KASLR on macOS for Apple silicon. Our attack demonstrates the capability to effectively circumvent KASLR, despite the high entropy characteristics of macOS for Apple silicon.

Second, our work targets the KASLR implementation with the kernel isolation enabled, which mitigates most of the attacks proposed by previous work. There are some works [11, 33, 40, 63] that provide attack on the kernel isolation enabled KASLR implementations. They use small kernel stub codes mapped into user-space page tables for their attack. However, our experiment shows that there is no such code space in macOS for Apple silicon, making these attacks infeasible. Unlike the previous work, SysBumps does not depend on these code stubs, but instead uses system calls to gain transient access to kernel addresses in kernel mode. As a result, our attack can successfully bypass KASLR on macOS for Apple silicon, where the kernel isolation is enabled.

Side-channel attacks on Apple silicon. Although not abundant, there have been a few studies that have uncovered side-channel vulnerabilities on Apple silicon processors. These vulnerabilities have been found at both the microarchitectural and architectural levels of the hardware.

Most of the previous work focuses on vulnerabilities in microarchitectural components in the processor, in particular, a cache on Apple silicon. Shusterman et al. [53] disclosed a cache side-channel vulnerability on an M1 CPU, demonstrating an attack that leaks the secret of a web browser. The S^2C [71] discloses a vulnerability in LoadLinked/Store-Conditional (LL/SC) instructions, and introduces a timerless cross-core cache attack by exploiting the vulnerability. Cronin et al. [14] explores the feasibility of creating a cache occupancy channel [54] on the M1 CPU and demonstrates a website fingerprinting attack that utilizes this channel. Augury [61] exploits a vulnerability in a data memory-dependent prefetcher (DMP), designed to prefetch irregular address patterns, to leak arbitrary data in the cache. Branch difference [26] showed that Spectre attacks are still effective on Apple silicon. iLeakage [32] also exploits the Spectre-v1 vulnerability [36] on the Apple silicon processor and demonstrates an attack that extracts secret information from a Safari browser.

There are other works that focus on vulnerabilities in the architectural features of Apple silicon processors. M1racle [1] demonstrated a cross-process covert channel attack by leveraging some implementation flaws in the system registers of the processor. Hot Pixels [57] introduced a hybrid attack utilizing software to measure physical properties in Dynamic Voltage Frequency Scaling (DVFS) to conduct a website fingerprinting attack on Apple Silicon.

Similar to our work, PACMAN [49] introduced a TLB-targeted attack on Apple silicon-based devices, specifically focusing on bypassing the ARM pointer authentication by exploiting the Spectre-v1 vulnerability. As discussed in Section 3.1, our work differs from PACMAN in that our primary goal is to break KASLR in macOS for Apple silicon, and we use a more advanced PMU-based methodology to reverse-engineer the internals of the TLB.

8 Conclusion

In this paper, we presented SysBumps, the first speculative execution attack aimed at circumventing KASLR within macOS for

Apple silicon. We introduced a novel technique that allows unprivileged attackers to transiently access kernel memory, despite the barrier of double map, the kernel isolation defenses implemented in macOS. With this technique, we built an attack primitive that enables distinguishing between valid and invalid kernel addresses by observing the TLB state, which is necessary for the KASLR break attack. We also performed reverse-engineering the hidden internals of the TLB on various M-series processors using a PMU, through which we successfully implemented the SysBumps attack. Throughout the evaluation, we demonstrated that SysBumps can effectively break KASLR across different M-series processors and macOS releases. We also proposed comprehensive mitigation strategies that include both software and hardware solutions. These mitigations can address the underlying vulnerabilities exposed by our attack and provide a more resilient defense against sophisticated attacks.

Acknowledgments

This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT) (No.2023R1A2C2006862).

References

[1] [n. d.]. *M1ssing Register Access Controls Leak EL0 State.* https://m1racles.com/

[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook. js: Attacking Chrome strict site isolation via speculative execution. In *2022 IEEE Symposium on Security and Privacy (S&P).* IEEE, 699–715.

[3] Apple. 2021. *Kernel extensions in macOS.* https://support.apple.com/fr-ml/guide/security/sec8e454101b/web

[4] Apple. 2023. *xnu/osfmk/arm/arm_init.c.* https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/arm/arm_init.c

[5] Apple. 2023. *xnu/osfmk/armx86_64/copyio.c.* https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/x86_64/copyio.c

[6] Apple. 2024. *xnu/bsd/kern/syscalls.master.* https://github.com/apple-oss-distributions/xnu/blob/main/bsd/kern/syscalls.master

[7] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22).* 971–988.

[8] 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.* 1046–1060.

[9] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of ROP Chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020).* 1–16.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 769–784.

[11] 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.* 481–493.

[12] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20).* 1093–1110.

[13] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 952–963.

[14] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An exploration of ARM system-level cache and GPU side channels. In *Proceedings of the 37th Annual Computer Security Applications Conference.* 784–795.

[15] Apple Developer. 2014. *Overview of the Mach-O Executable Format.* https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Article

[16] Apple Developer. 2024. *Apple Silicon CPU Optimization Guide.* https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 1–13.

[18] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. Lazarus: Practical side-channel resilient kernel-space randomization. In *20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2017).* 238–258.

[19] 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.* 1871–1885.

[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18).* 955–972.

[21] Daniel Gruss, Dave Hansen, and Brendan Gregg. 2018. Kernel isolation: From an academic idea to an efficient patch for every computer. *;login:* 43, 4 (2018), 10–14.

[22] 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: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9.* 161–176.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 368–379.

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15).* 897–912.

[25] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. 2024. Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation. In *2024 IEEE Symposium on Security and Privacy (S&P).* IEEE, 158–158.

[26] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different-Spectre Attacks on Apple Silicon. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* 116–135.

[27] Ralf Hund, Thorsten Holz, and Felix C Freiling. 2009. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *18th USENIX security symposium (USENIX Security 09).* 383–398.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy (S&P).* IEEE, 191–205.

[29] Intel. 2024. *Intel 64 and IA-32 Architectures Software Developer Manuals.* https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[30] Hyerean Jang and Youngjoo Shin. 2023. MicroCFI: Microarchitecture-Level Control-Flow Restrictions for Spectre Mitigation. *IEEE Access* (2023).

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 380–392.

[32] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.* 2038–2052.

[33] Suryeon Kim, Seungwon Shin, and Hyunwoo Choi. 2023. AVX-TSCHA: Leaking information through AVX extensions in commercial processors. *Computers & Security* 134 (2023), 103437.

[34] Taehun Kim, Hyeongjin Park, Seokmin Lee, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. 2023. DevIOus: Device-Driven Side-Channel Attacks on the IOMMU. In *2023 IEEE Symposium on Security and Privacy (S&P).* IEEE, 2288–2305.

[35] Taehun Kim and Youngjoo Shin. 2022. ThermalBleed: A Practical Thermal Side-Channel Attack. *IEEE Access* (2022).

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[37] 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).* USENIX Association.

[38] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (S&P).* IEEE, 39–53.

- [39] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 309–321.
- [40] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*. 643–660.
- [41] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*. 549–564.
- [42] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 355–371.
- [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 *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.
- [44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 605–622.
- [45] William Liu, Joseph Ravichandran, and Mengjia Yan. 2023. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*. 10–18.
- [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*. 2109–2122.
- [47] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. 2021. Bypassing memory safety mechanisms through speculative control flow hijacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 633–649.
- [48] Microsoft. 2018. *KVA Shadow: Mitigating Meltdown on Windows*. https://msrc.microsoft.com/blog/2018/03/kva-shadow-mitigating-meltdown-on-windows/
- [49] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 685–698.
- [50] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. In *ACM Transactions on Information and System Security (TISSEC)*. 1–34.
- [51] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753––768.
- [52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM SIGSAC Conference on Computer and Communications Security*. 298–307.
- [53] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium (USENIX Security 21)*. 2863–2880.
- [54] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. 639–656.
- [55] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 351–366.
- [56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 48–62.
- [57] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and Arm SoCs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6275–6292.
- [58] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shinn. 2022. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *2023 IEEE Symposium on Security and Privacy (S&P)*. 681–698. https://doi.org/10.1109/SP46214.2022.9833802
- [59] Tromer, Eran, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.
- [60] The UNIX and Linux Forums. 2005. *otool(1) [osx man page]*. https://www.unix.com/man-page/osx/1/otool/
- [61] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W Fletcher, and David Kohlbrenner. 2022. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1491–1505.
- [62] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2504–2519.
- [63] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*. 1415–1432.
- [64] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. 3825–3842.
- [65] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. 1187–1204.
- [66] Jidong Xiao, Hai Huang, and Haining Wang. 2015. Kernel data attack is a realistic security threat. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 135–154.
- [67] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 414–425.
- [68] YaoYuan. 2023. *XNU kperf/kpc demo*. https://gist.github.com/ibireme/173517c208c7dc333ba962c1f0d67d12
- [69] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. 719–732.
- [70] Tingting Yin, Zicong Gao, Zhenghang Xiao, Zheyu Ma, Min Zheng, and Chao Zhang. 2023. KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5039–5054.
- [71] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. 2023. Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1973–1990.

Table 7: List of exploitable system calls.

Syscall no.	Prototype	Description
5	`int access(const char *path, int amode)`	Check access permissions of a file or pathname
9	`int link(user_addr_t path, user_addr_t link)`	Make a hard file link
10	`int unlink(user_addr_t path)`	Remove directory entry
12	`int chdir(user_addr_t path)`	Change current working directory
15	`int chmod(user_addr_t path, int mode)`	Change mode of file
34	`int chflags(char *path, int flags)`	Set file flags
56	`int revoke(char *path)`	Revoke file access
57	`int symlink(char *path, char *link)`	Make symbolic link to a file
58	`int readlink(char *path, char *buf, int count)`	Read value of a symbolic link
136	`int mkdir(user_addr_t path, int mode)`	Make a directory file
165	`int quotactl(user_addr_t path, int mode)`	Manipulate filesystem quotas
188	`int stat(const char *path, int cmd, ... )`	Get file status
190	`int lstat(user_addr_t path, user_addr_t ub)`	Get file status
191	`int pathconf(char *path, int name)`	Get configurable pathname variables
200	`int truncate(char *path, off_t length)`	Truncate or extend a file to a specified length
220	`int getattrlist(const char *path, struct attrlist *alist, ... )`	Get file system attributes
221	`int setattrlist(const char *path, struct attrlist *alist, ... )`	Set file system attributes
223	`int exchangedata(const char *path1, const char *path2, ... )`	Atomically exchange data between two files
234	`user_ssize_t getxattr(user_addr_t path, user_addr_t attrname, ... )`	Get an extended attribute value
235	`user_ssize_t fgetxattr(int fd, user_addr_t attrname, ... )`	Get an extended attribute value
236	`int setxattr(user_addr_t path, user_addr_t attrname, ... )`	Set an extended attribute value
237	`int fsetxattr(int fd, user_addr_t attrname, ... )`	Set an extended attribute value
238	`int removexattr(user_addr_t path, user_addr_t attrname, ... )`	Remove an extended attribute value
239	`int fremovexattr(int fd, user_addr_t attrname, ... )`	Remove an extended attribute value
240	`user_ssize_t listxattr(user_addr_t path, user_addr_t namebuf, ... )`	List an extended attribute value

A Exploitable system calls list

Table 7 presents a list of system calls that are identified as potential attack vectors in Section 3.2. Arguments critical to executing SysBumps attack are underlined; these arguments permit the specification of user and kernel addresses, facilitating our attack. These arguments are typically of the 'char *' type or 'user_addr_t' type, representing a path or attribute name (attrname), which correspond to user-space memory addresses.