


Breaking KASLR on Mobile Devices without Any Use of Cache Memory

Milad Seddigh

 Cyberspace Research Institute,
Shahid Beheshti University, Iran
milladseddigh7@gmail.com

Mahdi Esfahani

 Department of Electrical
Engineering, Sharif University of
Technology, Tehran, Iran
m.esfahani@sharif.edu


Sarani Bhattacharya

 IMEC, Belgium
Sarani.Bhattacharya@imec.be

Mohammad Reza Aref

Department of Electrical Engineering,
Sharif University of Technology,
Tehran, Iran
aref@sharif.edu

Hadi Soleimany

 Cyberspace Research Institute,
Shahid Beheshti University, Iran
h_soleimany@sbu.ac.ir

ABSTRACT

Microarchitectural attacks utilize the performance optimization constructs that have been studied over decades in computer architecture research and show the vulnerability of such optimizations in a realistic framework. One such highly performance driven vulnerable construct is speculative execution. In this paper, we focus on the problem of breaking the kernel address-space layout randomization (KASLR) on modern mobile devices without using cache memory as a medium of observation. However, there are some challenges to breaking KASLR on ARM CPUs. The first challenge is that eviction strategies on ARM CPUs are slow, and the microarchitectural attacks exploiting the cache as a covert channel cannot be implemented on modern ARM CPUs. The second challenge is that non-canonical addresses are stored in the store buffer, although they are invalid. As a result, previous microarchitectural attacks distinguish such addresses as valid kernel addresses erroneously.

In this paper, we focus on these challenges to close current gaps in the implementation of recent attacks against modern CPUs. We show how a Translation Look-aside Buffer (TLB) can be used to circumvent the cache memory as a covert channel in order to attack ASLR on both ARM and Intel CPUs. To the best of our knowledge, we are the first to break KASLR on ARM-based Android and iOS mobile devices. Furthermore, our attacks can be performed in JavaScript to break KASLR of the browser without the need for an Evict+Reload operation, which consumes a lot of time. The results of our attacks show that the attacker can distinguish whether or not the virtual address is valid in less than 0.0417 seconds and 0.0488 seconds on Android and iOS mobile devices, respectively.

This paper has been accepted at ASHES 2022 and will be presented there.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM... \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

KEYWORDS

Speculative execution, Non-canonical addresses, KASLR

ACM Reference Format:

Milad Seddigh, Mahdi Esfahani, Sarani Bhattacharya, Mohammad Reza Aref, and Hadi Soleimany. 2022. Breaking KASLR on Mobile Devices without Any Use of Cache Memory. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Microarchitectural attacks exploiting the weaknesses of the speculative execution of modern CPUs have posed a serious threat to the security of computer systems. They have devastating impacts on operating systems, virtual machines, as well as cryptographic algorithms like AES hardware accelerators [22]. For instance, cache-based side-channel attacks [19] have been mounted on ARM-based mobile phones, showing that these devices have the same vulnerability as computer systems. As most cache attacks on ARM use the Evict+Reload technique to leak the victim's sensitive information, the researchers have focused on discovering the method for countering this technique. As a result, Williamson [30] proposed a method for preventing the eviction of an inclusive cache line in the cross-core attacks.

We have witnessed a variety of attacks in recent years, including Meltdown-like [6, 24, 25] and Spectre-like [8, 18, 27] attacks on the microarchitecture of processors, which make the internal state of the CPU visible to the attacker. In Meltdown-like attacks, an unprivileged attacker gains access to a kernel address that is inaccessible to the user. Subsequently, accessing that kernel address causes an exception that diverts the control flow to an exception handler. However, the CPU performs the subsequent transient instruction based on the secret value of that kernel address. Finally, the attacker reveals the secret value via a microarchitectural covert channel. In other words, Meltdown-like attacks violate any hardware isolation of virtual machines and do not rely on any software weakness.

On the other hand, KAISER was built to combat the Meltdown attack, which violates hardware isolation. By using KAISER, there is no kernel address mapped in the user space. In other words, the translation table related to kernel addresses is isolated from the

translation table relevant to user-space addresses. This countermeasure for Intel CPUs is now available in the latest releases of the Linux kernel, named Kernel Page Table Isolation (KPTI) [11, 13, 16]. Additionally, ARM developers provided FEAT_CSV3 [3], a countermeasure which recognizes whether or not a load operation from a kernel address can be used to leak private data.

There are some Meltdown-like attacks that exploit other internal components of the CPU, such as the store buffer and the line fill buffer. For instance, Fallout [22], a transient attack that recovers private information from the store buffer by exploiting the Write Transient Forwarding optimization, which forwards speculatively the store to the subsequent load with the same page offset. The RIDL attack [28] is a class of speculative-execution attacks that leaks arbitrary in-flight data with no assumptions from a microarchitectural component called the Line Fill Buffer that keeps track of outstanding memory requests and merges various in-flight stores. Also, another Meltdown-like attack is the Zombieload [25]. This attack targeting the line fill buffer can cause data leakage even on CPUs that are immune to all Meltdown-like attacks.

In recent years, we have observed several Meltdown-like attacks that aim to break KASLR from JavaScript [6, 24]. As JavaScript does not provide any flush operations, Meltdown-like attacks usually leverage some eviction strategies instead of flush operations in order to evict data from the cache memory. Also, JavaScript does not allow any access to the kernel addresses architecturally. Therefore, previous attacks usually access an out-of-bound accessible array instead of accessing the kernel address directly.

1.1 Challenges of Implementation of Microarchitectural Attacks on ARM and Intel CPUs

In this part, we discuss the probable difficulties associated with implementing microarchitectural attacks on ARM and Intel CPUs, and then demonstrate that our attack can circumvent these challenges. These challenges are as follows:

- (1) The vast majority of ARM CPUs lack a *flush* instruction. Additionally, existing eviction strategies are not very efficient. Therefore, Lipp et al. [19] addressed this issue by developing a fast eviction strategy that is also efficient on ARM CPUs. To perform this eviction strategy, the attacker must use `/proc/self/pagemap` to derive physical addresses.
- (2) After microarchitectural attacks targeting store-to-load forwarding feature have been presented by Google Project Zero, ARM developers proposed a countermeasure called Speculative Store Bypass Safe (SSBS). When this countermeasure is enabled on a specific ARM CPU, the hardware is not permitted to execute the store-to-load forwarding feature between a store and subsequent loads. This countermeasure is enabled for architectures from Armv8.0 to Armv8.4. As a result, Data Bounce and Fetch Bounce attacks [24], which exploit the store-to-load forwarding feature cannot be performed on ARM CPUs.
- (3) The architecture of some Intel CPUs for virtual address ranges is different from other CPUs. For instance, the official documentation in The Linux Kernel Archive [2] states that the bits 47 to 63 of non-canonical addresses may be

all '0' or '1', and such addresses in Intel CPUs range between `0x0000800000000000` and `0xffff7fffffff`, while previous attacks, such as [24] claim that the bits 47 to 63 of non-canonical addresses are not all '0' or '1'.

Also, non-canonical addresses behave differently to invalid kernel addresses, and although such addresses are invalid, they enter the store buffer. As a result, when the attacker performs the Data Bounce or the Fetch Bounce attacks [24] without any knowledge of virtual address ranges, they might erroneously recognize non-canonical addresses as valid addresses due to the fact that they enter the store buffer. In other words, the Data+Bounce and the Fetch+Bounce attacks [24] will fail in distinguishing the ranges between `0x0000800000000000` and `0x0000ffffffff`, and also between `0xffff000000000000` and `0xffff7fffffff`.

- (4) As JavaScript does not include a way to flush a specific address from the cache memory, previous works [10, 18, 23, 26] use Evict+Reload instead of Flush+Reload. In order to perform Evict+Reload, the attacker accesses an array that is bigger than the last-level cache to ensure that the targeted address will be evicted from the cache memory. The major disadvantage of using Evict+Reload is that the attacker would need to evict 256 sets, which consumes a lot of time and can lead to errors.

Although these challenges can make microarchitectural attacks difficult for the attacker, our attack can solve these challenges due to the fact that it does not exploit any cache memory and does not require any eviction operations.

1.2 Our Contribution

According to the microarchitecture of ARM-based mobile devices, the first access to a valid access results in a compulsory miss. But a subsequent access to the same valid access results in a TLB hit. Whereas this does not hold for the case of an invalid address, where repeated accesses to invalid access still results in TLB misses. Thus, a TLB miss that occurs for the second access to an invalid address can be used as a distinguishing observation between a valid and invalid address. Thus, the attacker can use the timing difference between a TLB hit and a TLB miss to determine whether a target address is valid.

In the first part of our two-fold contribution, we present a method that enables us to recognize valid and invalid kernel addresses on ARM-based Android and iOS mobile devices, which effectively derandomizes KASLR. Our first attack exploits the timing difference between a TLB hit and a TLB miss on ARM-based mobile devices. By leveraging the TLB and out-of-order optimization, our attack overcomes major challenges in performing microarchitectural attacks on ARM CPUs. In other words, our attack does not need any cache memory as a covert channel, which can circumvent eviction strategies. On the other hand, our attack does not require to exploit the store-to-load forwarding feature, which can lead to circumventing SSBS countermeasure. Also, our attack can be performed on Intel CPUs successfully.

One major limitation of the attacks targeting KASLR is that the attacker needs to find virtual address ranges on Intel CPUs. Because the architecture of Intel CPUs is different, and the bits 47 to 63 of

non-canonical addresses may be all '0' or '1'. For this reason, in the second part of our contribution, we propose another novel attack that derives virtual address ranges by determining whether a virtual address is canonical or non-canonical. Our attack leverages the timing difference between the execution time of the second access to non-canonical addresses and canonical addresses. Our technique is applicable to both Intel and ARM CPUs. Since for non-canonical addresses, there is a strange case that such addresses are entered into the store buffer, although they are invalid. Besides, there is no page table walk for such addresses, unlike canonical addresses. Therefore, Data Bounce and Fetch Bounce attacks [24] exploiting store buffer and cache memory are not able to identify non-canonical addresses without any knowledge of virtual addresses. As a result, the major advantage of our second contribution is that not only can it recognize virtual address ranges by finding non-canonical addresses, but it also does not require any cache memory as a covert channel. Although previous microarchitectural attacks from JavaScript would need Evict+reload operation which wastes much time, our attacks would not require any Evict+Reload operation and is faster than previous attacks from JavaScript.

1.3 Outline

Section 2 briefly explains the background of transient execution attacks. In Section 3, we present our new attacks for breaking ASLR and distinguishing canonical addresses from non-canonical addresses. In Section 4, we discuss the results of our experiments. In Section 5, we explain related works and related countermeasures. Finally, we conclude our paper in Section 6.

2 PRELIMINARIES

In this section, we provide the background needed for this paper.

2.1 Transient-execution Attacks

Modern processors employ an optimization technique called "out-of-order execution", which enables the CPU to process instructions in parallel rather than sequentially. When the execution unit of the current operation is running, subsequent execution units can be performed ahead. For this reason, modern processors begin by fetching and decoding instructions, which are divided into micro-operations (μop) that are placed in the Reorder Buffer (ROB). Besides, μops with operands require storage space that is allocated not only from the load buffer but also from the store buffer and from the register file. Furthermore, the CPU is able to schedule subsequent μop when the operation of μops being executed is unavailable. When a μop is available for execution, the scheduler schedules it for execution and stores the results in the relevant registers, load buffer entries, or store buffer entries. When the subsequent μop is completed, it is retired and its results are committed. Similarly, speculative execution causes the CPU to guess the outcome of a conditional branch in order to take the most likely path. When the CPU discovers that an instruction executed speculatively was executed incorrectly, the result is never committed and the reorder buffer and pipeline are flushed. However, this may have unintended consequences for the microarchitectural units such as cache and translation look aside buffer (TLB). As a result, instructions that are executed speculatively have side effects on cache memory and are referred to as

transient instructions. As a consequence of these side effects on the microarchitectural units, attacks known as "transient execution attacks" can be used by the attacker to disclose the private data of the victim. Typically, these attacks use a cache-based covert channel to extract private data from other security domains [7, 17, 18, 21].

When the execution unit must store data in memory, rather than waiting for the store to complete, data is queued in the store buffer to allow the execution unit to continue processing instructions from the current execution sequence. In other words, modern CPUs use store and load buffers to interact with cache memory. The load buffer includes the requests for data to be fetched from memory, whereas the store buffer includes requests for data to be stored in memory. Therefore, the store buffer prevents the CPU from stalling while waiting for the memory subsystem to complete the store and ensures that the results are stored in the memory in the correct sequence regardless of whether speculative execution is used. A store buffer entry is allocated only when a store operation is added to the reorder buffer. This entry in the store buffer needs both a physical address and a virtual address. While the store buffer improves store operation latency, it adds load complexity by requiring the load buffer to search the whole store buffer in parallel with the L1 cache for each load operation. When the store's entire address is equal to the entire address of the subsequent load, the data from the store buffer is directly transferred to the load buffer. This performance optimization is referred to as "store-to-load forwarding". Additionally, to accelerate the store-to-load forwarding operation, the CPU may assume that the address of a load matches the address of a preceding store only when the least significant 12 bits match. Also, when the processor incorrectly speculates that the data should be sent from the store buffer to the load buffer, the data is not committed and then reorder buffer and pipeline are flushed. This is referred to as "write transient forwarding" and can be exploited by the attacker to extract private data from the kernel memory [3, 22].

Data Bounce [24] makes use of the store buffer's feature in which the whole physical address is needed for a valid entry. While the ROB reserves the entry for store operation, store-to-load forwarding is only possible if the virtual address of the target load is valid. Hence, a virtual address that does not correspond to a valid physical address cannot be forwarded to subsequent loads. If Data Bounce is successful for a target address, the attacker will discover that the target address is valid and backed up with a physical address. Otherwise, the target address is invalid. As a result, Data Bounce can only recognize which addresses are valid and which are invalid, and the results of transient instruction are never architecturally committed.

2.2 Address Translation and Address Space Layout Randomization

In order to isolate processes from one another, modern processors support virtual memory as an isolation mechanism. Therefore, the processes use virtual addresses rather than physical addresses and are architecturally protected from one another. A multi-level page translation table is used to convert a virtual address to a physical address. Translation table base registers (TTBR0 for user space addresses and TTBR1 for kernel addresses) on ARM CPUs

indicate the location of the translation table. The translation table base control registers determine whether TTBR0 should be used or TTBR1. Because navigating translation tables for getting the physical address associated with a particular virtual address consumes considerable time, processors use smaller special caches called translation-lookaside buffers (TLBs) to store the physical address associated with each virtual address [3]. The Cortex-A57 processor contains a 2-level TLB structure (L1 and L2). The L1 instruction TLB implements a 48-entry fully-associative structure which caches entries of three different page sizes, i.e., 4KB, 64KB, and 1MB, of virtual address to physical address mappings. The L1 data TLB implements a 32-entry fully-associative TLB that is utilized for data loading and storing and caches entries of three different page sizes, i.e., 4KB, 64KB, and 1MB, of virtual address to physical address mappings. An instruction TLB hit takes only a single clock cycle to access the translation and then returns the related physical address to the instruction cache for comparison. L2, which implements a 1024-entry 4-way set-associative structure, handles misses from the L1 instruction and data TLBs, and also supports the page sizes of 4K, 64K, 1MB, and 16MB [1].

To obtain confidential information about virtual addresses, the attacker usually needs to have knowledge about the addresses of the targeted system. Various methods, such as address space layout randomization (ASLR), non-executable stacks, and stack canaries, can be used to thwart such attacks. For every boot, KASLR randomizes the offsets of all code, data, and so on. As a result, KASLR makes the implementation of some microarchitectural attacks more difficult.

3 METHODOLOGY

In this section, we build step-by-step the methodology for breaking KASLR in ARM and Intel processors. Then, we aim to find the virtual address ranges of modern CPUs by distinguishing non-canonical addresses. We start by defining the threat model for the adversary.

3.1 Threat Model

In our attacks on Intel CPUs, we consider a multi-user environment that is run on a Linux-based operating system, where several concurrent processes are sharing the same physical machine. Also, the attacker and the victim are running on the same core. The attacker who has user-level privileges in the system, does not need to know virtual addresses of the victim's system. The adversary observes the execution time with timestamp counters over a series of accesses to various memory elements with varying virtual address spaces.

3.2 Inception of the Attack

According to the microarchitecture of ARM and Intel CPUs, the first access to a valid target virtual address causes the corresponding physical address of the target virtual address to be cached in the TLB. For the consequent accesses to that same target address, a TLB hit occurs. However, in the case of an invalid virtual address, there is no corresponding physical address in the TLB. A TLB miss occurs for the second access to that target invalid address. Therefore, the attacker can distinguish whether a virtual address is valid or not by observing a timing difference between a TLB miss and a TLB hit.

Our first attack, which aims to break KASLR, exploits this behavior of TLB hit and miss in modern CPUs. Our observation is that the execution time of the second access to an invalid address is greater than to a valid address. Since for the first access (store operation) to a valid address, the related physical address is stored in the TLB and then enters the store buffer, while for the first access to an invalid address, no physical address is stored in the TLB. Thus, there is a time difference between a TLB miss and a TLB hit for the second access, which is executed out-of-order. In other words, for the second access (load operation) to a valid virtual address, the physical address is sent from TLB to the load buffer instead of walking translation tables. As a result, a TLB hit has occurred for the second access. But, for the second access to an invalid virtual address, the Memory Management Unit (MMU) always walks the translation tables, and the target virtual address is not backed by the physical address. As a consequence, the attacker can learn that a TLB miss has occurred for the second access. The steps of our first attack can be summarized as follows:

Step 1: The attacker first stores the private byte in the target virtual address.

Step 2: The attacker executes a transient instruction that is dependent on the target virtual address. Due to the out-of-order feature of modern CPUs, this transient instruction is executed in parallel with the preceding instruction, but its results will not be committed.

Step 3: The attacker measures the execution time of the store operation and transient instruction.

If the execution time is greater than the timing threshold between a TLB hit and a TLB miss, the attacker can interpret that the target virtual address is invalid due to the fact that the target virtual address is not in the TLB. However, when the execution time is smaller than the threshold, the attacker can learn that the target virtual address is valid since the target virtual address is in the TLB.

Our second attack is based on an observation in the ARM and Intel CPUs that the execution time of the second access to a non-canonical address is greater than a canonical address. Based on our extensive observations on a varied range of devices, we report for the first time the difference in access times for canonical and non-canonical addresses, where the accesses to non-canonical addresses are consistently higher than canonical addresses. For the second access to non-canonical virtual addresses, no virtual address is backed by a physical address, and no translation table walk occurs for such invalid addresses. On the contrary, for the second access to a canonical valid virtual address, the related physical address is sent from TLB to the load buffer. However, there is a third case as well. For the second access to a canonical invalid virtual address, a translation table walk occurs, and the target virtual address is not backed by a physical address.

In our second attack, the attacker first stores the private byte in the target virtual address and then executes a transient instruction that is dependent on the target virtual address. Later on, he measures the execution time of both accesses (store operation and transient instruction) to the target virtual address. As it is illustrated in Figure 1, there are three sets of timing ranges, and the attacker can choose the timing thresholds between these three sets appropriately. These three sets of timing ranges can be described as follows:

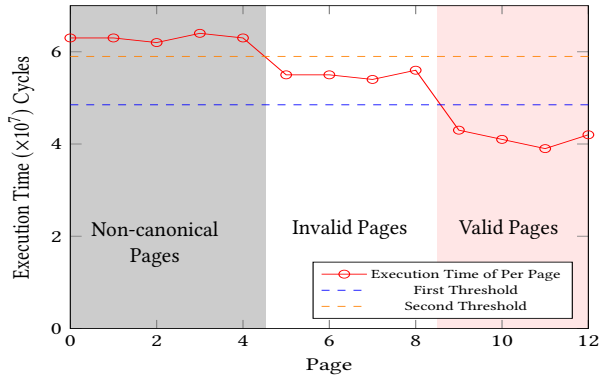


Figure 1: Our second attack shows that when the execution time is longer than the second threshold, the target pages are non-canonical. If the execution time is smaller than the first threshold, the target pages are valid. Otherwise, the target pages are invalid.

Non-canonical Address. According to our observation, if the execution time is greater than the second threshold, the attacker can interpret that the target virtual address is non-canonical.

Valid & Canonical Address. When the execution time is smaller than the first threshold, the attacker can learn that the target virtual address is not only valid, but also canonical, and the target virtual address is backed by a physical address.

Invalid & Canonical Address. When the execution time is between the first and second thresholds, the target virtual address is not only invalid, but also canonical, and the target virtual address is not backed by a physical address.

3.3 Breaking KASLR

In this part, we explain how our first attack, described in Section 3.2, can break KASLR on ARM-based mobile devices and Intel CPUs.

In order to break KASLR, we leverage the property in modern CPUs that a translation table entry that generates a permission fault may be held in the TLB [1, 3]. In other words, valid virtual addresses are held in the TLB, while invalid virtual addresses are not held in the TLB. Therefore, the attacker can determine whether or not a target virtual address is valid only by finding whether a TLB hit has occurred for the second access to the target virtual address or not.

When an attacker stores his private data in a valid kernel address, the MMU walks through translation tables to find the corresponding physical address, which it then inserts into the TLB. For the second access to that specific kernel address, the physical address is read from TLB instead of walking translation tables. When the attacker measures the execution times of both accesses to that kernel address, he finds that the execution time is smaller than the threshold value and that the target address is backed by a physical address. However, for all accesses to the invalid kernel address, MMU always walks through translation tables in search of the corresponding physical address, but it can never discover one. As a result, an exception occurs as no address is backed by a physical address. When the

attacker measures the execution time of both accesses to that kernel address, he learns that the execution time exceeds the threshold.

As it is illustrated in Algorithm 1, the attacker’s objective is to determine whether the kernel address of the victim is valid or not. The attacker is the one who performs the transient execution. For this purpose, an exception is generated by storing the private byte in the target kernel address (Line 6 of Algorithm 1) and then handling the exception via a signal handler. Following that, the attacker executes a transient instruction which is dependent on the content of the target kernel address (Line 7 of Algorithm 1). Not only does the attacker measure the execution time of store operations but also of transient instruction. If the execution time exceeds the threshold time, a TLB miss has occurred, the target kernel address is invalid, and the address is not backed by a physical page. Alternatively, if the execution time is less than the threshold time, a TLB hit has occurred, the target kernel address is valid, and the address is backed by a physical page.

Algorithm 1 Our first attack for breaking KASLR

```

1: #define Threshold_Tlb_Miss (81.69 × 106)
2: char *victim = (char*) 0xffff800000000000;
3: time1 ← start time;
4: signal (SIGSEGV, catch_segv);
5: if (sigsetjmp (jbuf, 1) == 0) then
6:   victim[0] = 'c';
7:   victim[0] + 1;
8: end if
9: time2 ← end time;
10: if (time2 - time1) < Threshold_Tlb_Miss then
11:   printf ("kernel address is valid");
12: else
13:   printf ("kernel address is invalid");
14: end if

```

3.4 Finding Virtual Address Ranges

In this part, we first state why the attacker needs to find virtual address ranges. Then, we explain how our second attack, described in Section 3.2 can find the virtual address ranges of different CPUs.

To break KASLR using the techniques proposed in the literature, the attacker must know the ranges of the virtual addresses such as user space, non-canonical, and kernel space. According to [24], there is a strange case for non-canonical addresses in which the bits 47 to 63 are not all ‘0’ or ‘1’. Despite the fact that such addresses are invalid and never refer to a physical address, they are backed by a physical address using Data Bounce and Fetch Bounce. In other words, Data Bounce and Fetch Bounce attacks [24] are incapable of discriminating non-canonical addresses from canonical addresses, since even invalid non-canonical addresses are entered into the store buffer. Also, [2, 22] shows that the bits 47 to 63 for non-canonical addresses may be all ‘0’ or ‘1’. Therefore, Data Bounce attack [24] that cannot distinguish non-canonical addresses will fail to break KASLR without any knowledge of virtual address ranges.

Our attack identifies non-canonical addresses by measuring the execution time of the accesses to target virtual addresses. In our attack, the attacker first executes a transient execution which is reliant on storing private data in a target virtual address and loading that target virtual address out-of-order. Then, he measures the execution time of transient execution. If the execution time is longer

than the second timing threshold, the target virtual address is non-canonical. Otherwise, the target address is canonical. After finding non-canonical addresses, the attacker can determine virtual address ranges such as the address range between user space, non-canonical and kernel space.

Algorithm 2 illustrates our method for determining if the target virtual address is non-canonical or canonical. The attacker performs a transient execution that is handled by a signal handler. For this purpose, the attacker stores the private byte in the target kernel address (Line 7 of Algorithm 2). Following that, a transient instruction which is reliant on the content of the target kernel address is executed out-of-order (Line 8 of Algorithm 2). Then, the attacker measures the execution time of the store operation and transient instruction. When the execution time is smaller than the first threshold time, the attacker can interpret that the virtual address is valid and canonical and is backed by a physical page. The main reason that the execution time is less than the first threshold time is that the first access to a valid address requires a page table walk, whereas the second access requires reading the target address from the TLB. However, when the execution time is between the first and second threshold times, the attacker can interpret that virtual address as invalid and canonical. This comes from the fact that for the first access to the target virtual address, MMU triggers a page walk and cannot find the relevant physical address. Hence, MMU triggers another page walk for the second access to that specific virtual address. But, when the execution time exceeds the second threshold time, the attacker learns that the target virtual address is non-canonical and invalid, and no address is backed by a physical address, although there is no page table walk for such addresses.

The results of this attack show that for ARM Cortex-A57 and ARM Cortex-A53, non-canonical address range is $0x0000ffffffffff - 0xffff000000000000$, while for Intel Core i5-4460U, non-canonical address range is $0x0000800000000000 - 0xffff7ffffffffff$.

Algorithm 2 Our second attack for finding virtual address ranges

```

1: #define Threshold1 (81.69 × 106)
2: #define Threshold2 (82.2 × 106)
3: char *victim = (char*) 0x1234800000000000;
4: time1 ← start time;
5: signal (SIGSEGV, catch_segv);
6: if (sigsetjmp (jbuf, 1) == 0) then
7:   victim [0] = 'c';
8:   victim [0] + 1;
9: end if
10: time2 ← end time;
11: if ((time2 - time1) < Threshold1) then
12:   printf ("kernel address is valid & canonical");
13: end if
14: if ((time2 - time1) > Threshold2) then
15:   printf ("kernel address is invalid & non-canonical");
16: end if
17: if (Threshold1 < (time2 - time1) < Threshold2) then
18:   printf ("kernel address is invalid & canonical");
19: end if

```

4 EXPERIMENTAL SETUP AND RESULTS

In this section, we first discuss the exception handling used for ARM and Intel processors. Then, we explain the methods we utilized to measure time in our attacks. Finally, we evaluate the performance

and accuracy of our attacks. Also, Our experimental codes are archived in ¹.

4.1 Exception Handler

When an attacker attempts to access a kernel address, it causes an exception, and the remainder of the program is terminated by default.

Generally, there are four distinct ways to handle this type of exception. The first is the fork-and-crash technique. A forked process executes the instruction, which results in an exception, and its parent resumes continuing execution after the exception occurs. The second is a signal handler that catches the exception and resumes execution. The third method is to suppress the exception and wrap the attack code in a mispredicted branch that speculatively resumes the attack. Finally, the exception can be suppressed by wrapping it in a hardware transaction. This efficient method is called Intel's Transactional Synchronization Extensions (TSX), and it consists of an x86-64 instruction set expansion that supports hardware transactions. However, this efficient method is not supported on ARM CPUs and is applicable only to Intel CPUs. As a result, we employed a signal handler to carry out our attacks on ARM. A signal handler is a function that the target environment calls if an exception occurs as a result of an attempt to access an unavailable address [20, 29]. Also, we employed TSX during the implementation of our attack on the Intel Core i5-4460U.

4.2 Experimental Setup

We analyzed our attacks on multiple modern CPUs, such as the Intel Core i5-4460U, the ARM Cortex-A53, and the ARM Cortex-A57.

Table 1 illustrates the features of ARM CPUs which we tested. The NVIDIA Jetson Nano Kit utilizes Linux Tegra 4.9.140 as its operating system and a quad-core ARM Cortex-A57 processor clocked at 1430 MHz. The Raspberry Pi 3 utilizes Raspbian as its operating system and a 64-bit quad-core ARM Cortex-A53 processor clocked at 1.2 GHz. The Samsung Galaxy J5 uses Android 5.1.1 Lollipop as its operating system and a 4-core ARM Cortex-A53 clocked at 1200 MHz. The Xiaomi Redmi Note 8 uses Android 9.0 Pie as its operating system and has a 4-core Cortex-A73 and a 4-core Cortex-A53 CPU clocked at 1.8 GHz. Also, the Huawei CAM-L21 uses Android 6.0 as its operating system and an octa-core Cortex-A53 clocked at 1200 MHz. The Apple iPhone 6, which we use for our attacks, utilizes iOS 12.5.5 as the operating system and a dual-core Typhoon (ARM v8-based) clocked at 1.4 GHz. The Intel Core i5-43460U used for our attacks is clocked at 3.2 GHz and uses Windows 10 as its operating system.

In order to perform our attacks on ARM-based mobile devices, we first deploy C4droid, which is one of the most powerful C/C++ IDEs for Android mobile devices. Later on, we run our C-implementation on the C4droid successfully. However, we use the C/C++ Program Compiler, which is an IDE for our implementations on iOS mobile devices.

4.3 Timing Measurements

While Intel x86 CPUs can utilize the unprivileged *rdtsc* instruction, which returns the value of the hardware timestamp counter for

¹<https://anonymous.4open.science/r/KASARM-0D2D/README.md>

Table 1: ARM CPUs used in this paper

Device	SoC	CPU
NVIDIA Jetson Nano Kit	Tegra X1	Cortex-A57
Raspberry Pi 3	Broadcom BCM2837	Cortex-A53
Samsung Galaxy J5	Qualcomm Snapdragon 410 MSM8916	Cortex-A53
Xiaomi Redmi Note 8	Qualcomm Snapdragon 665 SDM665	4-core Cortex-A73 and 4-core Cortex-A53
Huawei CAM-L21	HiSilicon Kirin 620 Chipset	Cortex-A53
Apple iPhone 6	Apple A8	ARMv8-A

time measurement, ARM CPUs do not support the *rdtsc* instruction. ARM CPUs instead use a performance monitoring unit called the cycle count register *PMCCNTR* that requires root privileges.

Lipp et al. [19] proposed three efficient timing sources that do not require root privileges. The first approach is to use the *PERF-COUNT-HW-CPU-CYCLES*, which is included in some versions of Android. The second method is the *POSIX* function *clock_gettime()*. The third method creates a time cycle counter by looping over an incremental variable.

We employed the second method (*clock_gettime()*) during the implementation of our attacks on ARM CPUs. Besides, we applied *rdtsc* instruction during the implementation of our attacks on the Intel Core i5-4460U.

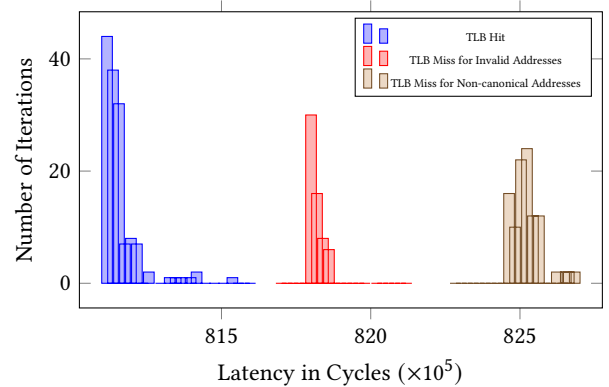
In order to find a threshold between TLB hit events and TLB miss events, we first need to measure the execution time of both accesses to different target virtual addresses in our first attack. Then, we can choose a threshold using the timing difference between TLB miss events and TLB hit events. Also, in order to mitigate the noise of the attack, we have to iterate our attacks numerous times. The number of iterations is different for each targeted CPU.

For instance, Figure 2 illustrates the execution time of both accesses to 200 virtual addresses of the NVIDIA Jetson Nano Kit by using *clock_gettime()*. Our results show that in 7×10^3 iterations of the attack, timing range for TLB hit events is 81.15×10^6 cycles - 81.67×10^6 cycles, whereas timing range for invalid addresses is 81.7×10^6 cycles - 82.12×10^6 cycles, and for non-canonical addresses is 82.23×10^6 cycles - 82.63×10^6 cycles. As a result, the threshold between a TLB hit and a TLB miss is 81.69×10^6 cycles (around 0.0571 seconds), while the threshold between the execution time of both accesses to an invalid address and a non-canonical address is 82.2×10^6 cycles (around 0.0574 seconds).

Also, we repeated these operations on various CPUs to find their timing thresholds. As it is illustrated in Table 2, for 5×10^3 iterations, the threshold between a TLB hit and a TLB miss for our first attack on devices using ARM Cortex-A53 clocked at 1200 MHz is 50.04×10^6 cycles (around 0.0417 seconds). But, this threshold for the Xiaomi Redmi Note 8 (in 7.2×10^3 iterations) and the Apple iPhone 6 (in 6×10^3 iterations) is 106.2×10^6 cycles and 68.32×10^6 cycles, respectively.

Besides, the threshold between the execution time of both accesses to an invalid address and a non-canonical address for the Apple iPhone 6 is 69.02×10^6 cycles (around 0.0492 seconds). However, this threshold for our second attack on devices using ARM Cortex-A53 clocked at 1200 MHz and 1800 MHz is 50.28×10^6 cycles (around 0.0419 seconds) and 107.1×10^6 cycles (around 0.0595 seconds), respectively.

Also, the threshold between a TLB hit and a TLB miss for 3×10^3 iterations of our first attack on Intel Core i5-4460U is 78.72×10^6


Figure 2: Timing measurements of our attacks on ARM Cortex A-57

cycles (around 0.0246 seconds). However, the threshold between the execution time of both accesses to an invalid address and a non-canonical address for 3×10^3 iterations of our second attack on Intel Core i5-4460U is 79.68×10^6 cycles (around 0.0249 seconds).

Table 2: Timing thresholds of our attacks for different CPUs

Device	CPU	Threshold between a TLB hit and a TLB miss	Threshold between a canonical and a non-canonical address
Raspberry Pi 3	ARM Cortex-A53	50.04×10^6 cycles	50.28×10^6 cycles
Samsung Galaxy J5	ARM Cortex-A53	50.04×10^6 cycles	50.28×10^6 cycles
Redmi Note 8	ARM Cortex-A53 & ARM Cortex-A73	106.2×10^6 cycles	107.1×10^6 cycles
Apple iPhone 6	ARMv8-A	68.32×10^6 cycles	69.02×10^6 cycles
Huawei CAM-L21	ARM Cortex-A53	50.04×10^6 cycles	50.28×10^6 cycles
Computer (ASUS)	Intel Core i5-4460U	78.72×10^6 cycles	79.68×10^6 cycles

4.4 Payload of our Attack Scenarios

In this section, we show that our attacks mentioned in Section 3 can break ASLR on Android and iOS mobile devices. First, we analyze the performance of our first attack on four different locations in the kernel memory: the kernel logical memory map, the Direct-Physical Map, the Kernel Text Segment, and kernel modules. Then, we discuss the results of our second attack on finding the virtual address ranges of modern CPUs. Also, in this section, we show the execution time of our attacks on different CPUs. To obtain the execution time of our attacks, we repeated each experiment several times to reduce the noise of the attack. The number of iterations of our attacks on each CPU is different.

4.4.1 De-randomizing the Direct-Physical Map and the Kernel Text Segment. The Linux kernel maps the entire physical memory into the kernel’s virtual address space by using a direct-physical map. To avoid microarchitectural attacks that require knowledge of kernel addresses, the map is located at a random offset within a defined range for each boot. However, our attack can find the offset of the direct-physical map and the kernel text segment of the Intel Core i5-4460U in around 0.0246 seconds for 3×10^3 iterations, according to Table 3. Also, our results on the Intel Core i5-4460 demonstrate that the address range for the direct-physical-map and the kernel text segment is $0xffff888000000000 - 0xffffc87ffffffffff$, and $0xffffffffff80000000 - 0xffffffffff9ffffffffff$, respectively.

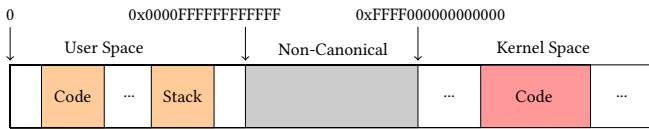


Figure 3: ARM Virtual Memory Layout

4.4.2 De-randomizing Kernel Logical Memory Map. Our attack’s primary objective is to retrieve the location of the kernel logical memory map on ARM CPUs. The results of our attack show that the address range for the kernel logical memory map is $0xffff000000000000 - 0xffff7fffffff$, i.e., a 128 TB region. Therefore, according to Table 3, we were able to extract the offset in less than 0.0571 seconds on the ARM Cortex-A57 and less than 0.0417 seconds on the Raspberry Pi 3, Samsung J5, and Huawei CAM-L21. Also, our attack can derive the offset in less than 0.0488 seconds on the Apple iPhone 6 for 6×10^3 iterations.

4.4.3 De-randomizing Kernel Modules. The results of our first attack show that the address range for kernel modules of ARM Cortex-A53 and Cortex-A57 is $0xffff800008000000 - 0xffff80000fffffff$, while the kernel modules of Intel Core i5-4460U range $0xfffff9a0000000 - 0xfffff9ffffff$. When we use our technique to recover kernel modules, we can determine the beginning and end of a module. Although `/proc/modules` can be used to extract information about kernel modules, it requires root access to provide the attacker with knowledge about module addresses. Thus, our first attack can identify the address where the module starts and ends.

4.4.4 Finding Virtual Address Ranges. The results of our second attack demonstrate that for ARM Cortex-A57 and ARM Cortex-A53, the user address range is $0x0000000000000000 - 0x0000ffffffff$, non-canonical range is $0x0000ffffffff - 0xffff000000000000$ and kernel address range is $0xffff000000000000 - 0xffffdfffffff$, as it can be seen in Figure 3. Also, our results on the Intel Core i5-4460U illustrate that the user address range is $0x0000000000000000 - 0x00007fffffff$, non-canonical range is $0x0000800000000000 - 0xffff7fffffff$ and the kernel address range is $0xffff800000000000 - 0xfffff9ffffff$ (Figure 4).

Also, according to Table 3, our second attack can distinguish a non-canonical address on ARM Cortex-A57, A53, and Intel Core i5-4460U in less than 0.0574 seconds, 0.0419 seconds, and 0.0249 seconds, respectively. Moreover, our second attack can find non-canonical addresses in less than 0.0492 seconds on the Apple iPhone 6.

Table 3: Experimental Results

Device	CPU	Time of Breaking KASLR	Time of Finding Virtual Address Ranges
NVIDIA Jetson Nano Kit	ARM Cortex-A57	0.0571s	0.0574s
Raspberry Pi 3	ARM Cortex-A53	0.0417s	0.0419s
Samsung Galaxy J5	ARM Cortex-A53	0.0417s	0.0419s
Redmi Note 8	ARM Cortex-A53 & ARM Cortex-A73	0.0590s	0.0595s
Apple iPhone 6	ARMv8-A	0.0488s	0.0492s
Huawei CAM-L21	ARM Cortex-A53	0.0417s	0.0419s
Computer (ASUS)	Intel Core i5-4460U	0.0246s	0.0249s

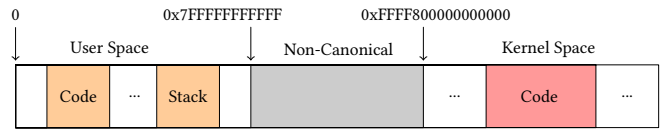


Figure 4: Intel Virtual Memory Layout

Also, we show the effect of microarchitectural attacks on ARMv8-A CPUs in Table 4. [5] states that Meltdown attack [20] cannot be performed on ARMv8-A CPUs because of their microarchitecture. Also, Data Bounce and Fetch Bounce attacks [24] cannot be implemented on ARMv8-A CPUs due to *FEAT_SSBS* countermeasure. However, our attack can be performed on ARMv8-A CPUs. The primary reason why our attack is applicable to this family of ARM CPUs is that kernel translation table isolation is optional in ARMv8-A CPUs and is mandatory in ARMv8.5A CPUs. Also, Table 4 shows that all important microarchitectural attacks can be performed on Intel CPUs. However, when KPTI is enabled, the attacker is not able to perform microarchitectural attacks on Intel CPUs.

Table 4: Effect of microarchitectural attacks on different ARM CPUs

Core	Meltdown [20]	Data Bounce [24]	Fetch Bounce [24]	Our attack
ARMv8-A:				
Cortex-A53	✗	✗	✗	✓
Cortex-A57	✗	✗	✗	✓
Cortex-A72	✗	✗	✗	✓
Cortex-A73	✗	✗	✗	✓
Intel:				
Intel Core i5-4460U	✓	✓	✓	✓
Intel Core i7-1165G7	✓	✓	✓	✓

4.4.5 Breaking ASLR from JavaScript. Our attacks can be used not only from unprivileged native applications but also from JavaScript to de-randomize KASLR in the browser. We analyze the performance of our attacks from JavaScript running in two browsers. Our analysis was carried out on Google Chrome 103.0.5060.66 (64-bit) and Microsoft Edge 103.0.1264.44 (64-bit).

There are some challenges for performing previous microarchitectural attacks [10, 18, 23, 26] in JavaScript. First, there is no timer with high accuracy in JavaScript. Second, because JavaScript lacks a flush instruction, previous attacks must use `Evict+Reload` rather than `Flush+Reload`. However, as our attacks do not require any cache memory as a covert channel, the second challenge of previous attacks is circumvented. The third challenge is that there is no access to kernel addresses architecturally in JavaScript.

Timing Measurement. In order to obtain timing measurements of our attacks, we can use a combination of a counting thread and `SharedArrayBuffers`, which has been re-enabled in Google Chrome. In comparison to the prior attacks [10, 26] which utilized `Uint32Array`, we can also utilize `BigUint64Array` to make sure that the counting thread does not overflow. Furthermore, as some browsers reduced the accuracy of the `Performance.now()` function, we can still use this function. By iterating our attacks numerous times, we can mitigate the noise of `Performance.now()`.

Covert Channel. As JavaScript does not include any flush operations, the attacker is not able to use Flush+Reload as a covert channel. As a result, previous attacks [10, 18, 23, 26] used Evict+Reload instead of the Flush+Reload operation. In Evict+Reload, the attacker accesses an array bigger than the last level cache to make sure that the target address will be evicted from the cache memory. The disadvantage of the use of Evict+Reload is that it can waste time and can cause errors. However, our attacks do not require any Evict+Reload operations since we do not exploit any cache memory as a covert channel.

Illegal Access. JavaScript does not allow any access to the kernel addresses architecturally. Therefore, we have to use the way Kocher et al. proposed in [18] to gain access to an out-of-bound offset of the array transiently. According to Kocher’s way, we repeat our attack over the virtual memory (relative to the accessible array) to distinguish whether the target virtual address is valid or not.

Prior microarchitectural attacks [10, 18, 23, 26] in the JavaScript environment are slower than the native environment due to slow Evict+Reload operations. However, the speed of our attacks in the JavaScript implementation is the same as the native implementation because we do not use any Evict+Reload operations in our attacks. As a result, the speed of our attacks in JavaScript is higher than prior microarchitectural attacks.

5 RELATED WORKS & COUNTERMEASURES

In this section, we first compare our attacks with previous side-channel attacks that broke KASLR. Then, we show that there are various countermeasures for mitigating previous attacks, but our attacks can circumvent these countermeasures.

5.1 Comparison to other Related Side-Channel Attacks on ASLR

Hund et al. [14] proposed three timing side-channel attacks to break ASLR. To carry out the first attack, it is necessary to keep track of the cache collisions with kernel addresses. This attack is not possible on mobile devices due to the existence of slow eviction strategies. The second attack makes use of double page faults and measures the difference in timing between a TLB miss and a TLB hit. The major drawback of this attack is that the attacker must employ an OS exception handler for each access to a kernel address, which decreases the speed of the attack, especially when this attack is performed in the cross-VM setting. For the third attack, the attacker leverages the timing differences of page faults caused by the TLB and address translation caches. Due to the attack’s reliance on evicting cache lines, eviction strategies in ARM processors mitigate it. In contrast to Hund’s attack [14, 15], our attack does not require any OS page fault handler. Furthermore, Hund’s attack requires two exception handlers for both accesses to a target kernel address, whereas our attack requires just one exception handler for both accesses to a target kernel address, which speeds up the attack process. Also, our attack makes use of an optimization known as out-of-order execution. By out-of-order execution, when the attacker accesses a target kernel address, subsequent instruction which accesses that target kernel address for the second time is executed partially, but is not retired. Gruss et al. [12] proposed a prefetch side-channel attack that exploits weaknesses in prefetch instructions and

is dependent on cache eviction. Additionally, this attack relies on evicting cache lines, which is not possible on mobile devices using ARM CPUs due to the slow eviction strategies. Evtvushkin et al. [9] presented attacks that need knowledge of the branch target buffer, which is not available on later CPUs than Haswell. Their attack relies on reverse engineering in order to obtain BTB addresses. Jang et al. [15] exploited Intel TSX and mitigated the noise of Hund’s attack [14] for breaking KASLR. This attack is dependent on Intel TSX and cannot be performed on processors manufactured prior to 2013. In addition, this attack is not applicable to ARM CPUs as TSX is not supported by ARM CPUs. Gruss et al. [24] presented the Data Bounce attack to defeat KASLR using the store buffer and a covert channel such as Flush+Reload. A Data Bounce attack cannot be performed on ARM CPUs due to eviction strategies. Additionally, this attack can be prevented by using the *SSBS* countermeasure on the newest ARM processors [3]. In the same work [24], Gruss et al. proposed the Fetch Bounce attack, which uses a combination of store buffer and TLB. As illustrated in Figure 4 of [24], the Fetch Bounce attack also uses the Flush+Reload attack to break KASLR. As a result, the Fetch Bounce attack cannot be performed on ARM-based mobile devices because of the eviction strategies and *SSBS* countermeasure. Canella et al. [6] showed that the Echoload attack can recognize load stalls from transiently executed loads. Although this attack detects valid from invalid virtual addresses, it needs the Flush+Reload technique, which cannot be carried out on most ARM CPUs.

Our attack does not need any cache memory as a covert channel. As shown in Table 5, our approach is the sole microarchitectural attack that can be carried out on Android and iOS mobile devices and requires only 0.0571 seconds to find whether the virtual address of the ARM Cortex-A57 is valid or not. In addition, our attack imposes no prerequisites, and hence all countermeasures designed to defend ARM CPUs from earlier microarchitectural attacks will fail.

Table 5: Comparison of timing side-channel attacks on ASLR of ARM CPUs

Attack	ARM CPUs	Time	Requirements
Hund et al. [14]	✗	-	-
Gruss et al. [12]	✗	-	Cache Eviction
Jang et al. [15]	✗	-	Intel TSX
Evtvushkin et al. [9]	✗	-	BTB reverse engineering
Data Bounce et al. [24]	✗	-	Evict+Reload
Our attack on ARM Cortex-A57	✓	0.0571s	-

5.2 Discussion on Ineffectiveness of Existing Countermeasures for our Proposed Attacks

Several countermeasures have been developed or implemented to mitigate known microarchitectural attacks on both ARM and Intel CPUs. In this part, we first briefly discuss the impact of these countermeasures on our attack. Also, we show that these countermeasures cannot protect ARM and Intel CPUs from our attacks.

Kernel Translation Table Isolation: Separating the kernel translation table from the user translation table is the best approach to defend against microarchitectural attacks on ARM. Kernel addresses are not mapped into a user translation table when this

strategy is used, resulting in the failure of most microarchitectural attacks. ARM processors have a countermeasure known as *FEAT_CSV3* [3]. *FEAT_CSV3* adds a mechanism for determining whether or not data loaded under speculation with a permission or domain fault can be used to form an address. This feature is optional in ARMv8.0 (Cortex-A72 and -A73) and is mandatory for ARMv8.5. Also, [4] notes that *FEAT_CSV3* has been implemented in the latest releases of Cortex-A75, -A76, -A77, and -A78. Therefore, our attack is applicable to the most recent releases of Cortex-A53, -A57, -A72, and -A73.

Speculative Store Bypass Safe: After researchers at Microsoft Security Response Center and Google Project Zero presented Spectre attack variant 4 in 2018, ARM developers devised a countermeasure called "speculative store bypass safe" (*FEAT_SSBS*) which indicates if a speculative store or load operation can result in the leakage of personal information via cache timing attacks. For ARM architectures using *FEAT_SSBS*, *PSTATE.SSBS* is a control that can be set by software to determine whether hardware is allowed to execute the load instruction, which has the same virtual address as the latest store instruction.

When the value of *PSTATE.SSBS* is set to 0, hardware is not permitted to load or store speculatively, whereas when the value of *PSTATE.SSBS* is 1, hardware is permitted to load or store speculatively. Software written for architectures from ARMv8.0 to ARMv8.4 will set *PSTATE.SSBS* to 0. In other words, hardware cannot be allowed to load or store speculatively in these architectures. It's worth noting that this mitigation cannot protect ARM CPUs against our attack because our attack does not exploit the store-to-load forwarding feature, whereas *FEAT_SSBS* mitigates the attacks leveraging store-to-forwarding optimization [3].

Speculative Store Bypass Barrier Instruction: The Speculative Store Bypass Barrier (*SSBB*) instruction is a memory barrier that protects ARM CPUs against write transient forwarding attacks. When a load operation to a specific address appears in program order after the *SSBB* instruction, and a store operation occurs before the *SSBB* instruction, then no store-to-load forwarding is executed between the store buffer and the load buffer. Also, the *SSBB* instruction is ineffective against our attack since our attack does not exploit the store-to-load forwarding capability of modern CPUs [3].

6 CONCLUSION

In this paper, we proposed a microarchitectural attack that can be leveraged to break KASLR on not only Android and iOS mobile devices, but also Intel CPUs. Our attack relies on measuring the execution time of the transient instruction and does not exploit any cache memory as a covert channel. Furthermore, our attack targets TLB and does not need any root access to derive virtual address ranges. The results of our attack demonstrate that the attacker can find whether the target virtual address is valid in less than 0.0417 seconds and 0.0488 seconds on ARM-based Android and iOS mobile devices, respectively. Also, we show that our attacks can be used from JavaScript to break ASLR in modern browsers. Consequently, we stress that isolation between the kernel translation table and the user translation table should be enabled for operating systems to prevent breaking KASLR by the attackers.

REFERENCES

- [1] 2017. ARM LIMITED. Cortex-A57 MPCore Processor Revision: r1p3 Technical Reference Manual. <http://infocenter.arm.com/help/topic/com>.
- [2] 2019. Linux. Complete virtual memory map with 4-level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.
- [3] 2021. Arm Architecture Reference Manual Armv8, for A-profile architecture. <https://developer.arm.com/documentation/ddi0487/gb/>.
- [4] 2021. Armv8.5-A CPU Updates - Arm Developer. <https://developer.arm.com>.
- [5] 2021. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. <https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability>.
- [6] 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.
- [7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtvushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 249–266.
- [8] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.
- [9] Dmitry Evtvushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 40:1–40:13. <https://doi.org/10.1109/MICRO.2016.7783743>
- [10] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, Vol. 17. 26.
- [11] Daniel Gruss, Dave Hansen, and Brendan Gregg. 2018. Kernel isolation: From an academic idea to an efficient patch for every computer. ; *login: the USENIX Magazine* 43, 4 (2018), 10–14.
- [12] 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, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 368–379. <https://doi.org/10.1145/2976749.2978356>
- [13] D Hansen. 2017. KAISER: unmap most of the kernel from userspace page table. *Linux Kernel Mailing List* (2017).
- [14] 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, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 191–205. <https://doi.org/10.1109/SP.2013.23>
- [15] 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, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 380–392. <https://doi.org/10.1145/2976749.2978321>
- [16] K Johnson. 2018. KVA Shadow: Mitigating Meltdown on Windows.
- [17] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).
- [18] Esmail 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)*.
- [19] 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, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 549–564. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>
- [20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56. <https://doi.org/10.1145/3357033>
- [21] 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.
- [22] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space. *CoRR* abs/1905.12701 (2019). [arXiv:1905.12701](http://arxiv.org/abs/1905.12701) <http://arxiv.org/abs/1905.12701>
- [23] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1406–1418.

- [24] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *CoRR* abs/1905.05725 (2019). arXiv:1905.05725 <http://arxiv.org/abs/1905.05725>
- [25] 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, CCS 2019, London, UK, November 11-15, 2019*. Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 753–768. <https://doi.org/10.1145/3319535.3354252>
- [26] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*. Springer, 247–267.
- [27] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11735)*, Kazuo Sako, Steve A. Schneider, and Peter Y. A. Ryan (Eds.). Springer, 279–299. https://doi.org/10.1007/978-3-030-29959-0_14
- [28] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. (2019), 88–105. <https://doi.org/10.1109/SP.2019.00087>
- [29] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. (2018).
- [30] WILLIAMSON. 2012. Line allocation in multi-level hierarchical data stores. Patent US8271733 B2, ARM Limited. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 1075–1091.