SnapMem: Hardware/Software Cooperative Memory Resistant to Cache-Related Attacks on ARM-FPGA Embedded SoC

Jingquan Ge and Fengwei Zhang, *Senior Member, IEEE*

***Abstract*—ARM-FPGA embedded SoCs have been widely used in the fields of 5G Wireless, next-generation ADAS (Advanced Driver-Assistance Systems) and Industrial Internet-of-Things due to its high performance and hardware design flexibility. However, this type of SoC suffers various security threats, one of which is cross-domain cache-related attacks, such as Flush+Reload, Flush+Flush, Meltdown and Spectre. Many hardware and software defenses have been proposed to resist these cross-domain cache-related attacks. However, hardware defenses require modifications of basic architecture, which cannot be deployed on existing devices. On the other hand, software runtime defenses have incomplete coverage or introduce significant performance overhead. In this paper, we propose SnapMem, a hardware/software cooperative memory that can make sensitive data burn after reading on ARM-FPGA embedded SoC. Any process can only access the SnapMem created by itself. Through the cooperation of software and hardware, SnapMem can transfer sensitive data in or out of main memory in real time. Based on this burn-after-reading mechanism, SnapMem can effectively prevent attackers from stealing sensitive data of the victim process or kernel space. Security and performance evaluations show that SnapMem can resist all cross-domain cache-related attacks while introducing lower performance overhead than other software runtime defenses on ARM-FPGA embedded SoC.**

***Index Terms*—Cache Attack, Defence, Memory, Cross-Domain, ARM-FPGA, Burn-After-Reading.**

I. Introduction

In recent years, with the rapid development of automotive electronic systems, Internet of Things and 5G Wireless, the market demand for SoCs with high performance, low power consumption and versatility is increasing. ARM-FPGA embedded SoCs, which combine both software and hardware, give system architects and ARM developers a flexible platform to meet the performance, power and functional needs of customers. This type of SoCs such as Xilinx Zynq and Versal series [1], have been widely used in the fields mentioned above. However, like Intel, AMD's x86 or Qualcomm, Samsung's ARM product, ARM-FPGA embedded SoC are also suffering a variety of security threats. Among them, cache-related attack is one of the most attractive threats.

Manuscript received 28 November 2023; revised 1 March 2024; accepted 5 April 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62372218; and in part by the Shenzhen Science and Technology Program under Grant SGDX20201103095408029. *(Corresponding author: Fengwei Zhang)*

Jingquan Ge and Fengwei Zhang are with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: gerty1986823@126.com; zhangfw@sustech.edu.cn).

Since the concept of cache-related attack was first proposed by Kocher [2] and Kelsey *et al.* [3], more types of cache-related attacks [4]–[14] have been presented by researchers. It becomes an important security threat to operating systems running on modern processors. Before 2018, since most cache-related attacks can only attack encryption implementations or shared pages, the security threats caused by them are limited. However, with the continuous emergence of Meltdown [15], Spectre [16] and their variants [17]–[25], the ability of cache-related attacks has been greatly enhanced. These cache-related attacks can break the isolation boundaries between different processes, or even between user and kernel spaces. More importantly, the target of these attacks is not limited to encryption implementations and shared pages, but extends to the entire process or kernel memory.

Researchers have proposed many defense solutions to detect and defend against cache-related attacks. However, these defenses have the following shortcomings. First, hardware defenses [26]–[33] need to modify the basic CPU architectures, which cannot be deployed on existing devices. Second, each software runtime defense [34]–[39] only targets a certain type of cache-related attack and cannot cover all. Third, significant performance loss may be introduced by some defense solutions [35], [37]–[40]. Therefore, designing defenses that can avoid the above three shortcomings has become a focus of research in both academia and industry.

On the other hand, the main goal of cache-related attacks is to steal sensitive data in main memory. Meanwhile, most of the data in main memory is not sensitive data and does not need to be protected. Therefore, not all data in main memory needs to be protected. The security of the system can be guaranteed as long as the sensitive data in the main memory can be protected. However, since cache-related attacks can theoretically steal the entire kernel and process memory, they can ideally steal all data in main memory. So a secure idea is to store sensitive data out of main memory. But this idea cannot be implemented on general SoC platforms because general SoCs can only access and process data in main memory. Fortunately, ARM-FPGA embedded SoC provides a platform for system developers to partially modify the hardware layer. We can freely design hardware peripherals on this type of SoC and mount them on AXI bus of ARM CPU. Using this software/hardware cooperative mechanism, we can design a more secure memory mechanism to store sensitive data out of main memory.

In this paper, we present SnapMem, a memory mechanism

that burns after reading. SnapMem is a software/hardware co-design, which consists of a hardware module and a software module on ARM-FPGA embedded SoC. The software module has two functions, the first is the management of process access permissions, and the second is the control of the hardware module. Each process can create its own unique SnapMem and can only access its own SnapMem. Any process that wants to access the SnapMem of another process is strictly prohibited. On the other hand, the software module controls the hardware module to move sensitive data from the main memory to the hardware memory or vice versa. When sensitive data is not being accessed, it is not stored in the main memory but in SnapMem's hardware memory. This sensitive data is only moved to the main memory when it is accessed using SnapMem's API . More importantly, SnapMem's hardware memory does not have an address map for ARM CPU, it can only be accessed by SnapMem's hardware module. Therefore, cache-related attacks are powerless against sensitive data in SnapMem.

We implement SnapMem by designing the software module, the hardware module, the SnapMem's APIs and modifying the Linux kernel. We perform various types of cache-related attacks to evaluate the security of SnapMem. It shows that SnapMem can defend against all cache-related attacks, which is more secure than other software defense solutions on ARM-based platforms. In addition, we conduct performance evaluations on SnapMem APIs and Linux original APIs, respectively. Moreover, we evaluate the system performance overhead of SnapMem by running UnixBench and SPECrate2017. The two overheads of SnapMem (0.07% on UnixBench and 0.04% on SPECrate2017) are better than all other software runtime defense schemes.

Our main contributions are summarized as follows:

- We create a hardware/software co-design of a more secure memory mechanism. This memory mechanism not only isolates sensitive data from main memory, but also isolates sensitive data from different processes, which can resist all cache-related attacks.
- Our design requires only one software module, one hardware module and minimal modification to the Linux kernel, which does not require modification of the basic CPU architecture. Therefore, it can be easily deployed on ARM-FPGA embedded devices that have been widely used.
- We conduct security and performance evaluations on the real ARM-FPGA embedded SoC. The security results show that the defense capability of this memory mechanism can cover all cache-related attacks. Performance evaluations indicate that our design has lower performance overhead than all other software runtime defenses.
- We propose a new idea for the secure improvement of memory mechanism. This memory mechanism only requires isolated control and memory modules to be mounted on the data and address buses, so it can be widely used on various platforms such as ASICs and CPU-FPGAs.

II. Background and Motivation

In this section, we first give the detailed descriptions of the ARM-FPGA embedded SoC. Then, we discuss the evolution of cache-related attacks. Finally, we will introduce the existing defenses and their shortcomings in depth.

A. ARM-FPGA embedded SoC

ARM-FPGA embedded SoCs combine the software programmability of ARM processors and hardware programmability of FPGAs, which enable differentiated and customizable solutions. This type of SoCs [1], [41] have flooded the market and are widely used in areas such as autonomous driving, 5G communications, and the Internet of Things. Among these SoCs, Zynq UltraScale+ MPSoC [42] is a typical representative. For simplicity without loss of generality, we take Zynq UltraScale+ MPSoC as an example to explain the hardware architecture of ARM-FPGA embedded SoC in detail.

Figure 1 shows the Simplified hardware architecture of ZU9EG, a representative product of the Zynq UltraScale+ MPSoC family. To be more intuitive, Figure 1 omits hardware components that are not related to our design, including dual-core Arm Cortex-R5F and on-chip memory, etc. As can be seen from Figure 1, the simplified hardware architecture of ZU9EG consists of three parts, namely quad-core Arm Cortex-A53 MPCore, the main memory and FPGA. On Zynq UltraScale+ MPSoC, the communication between the ARM MPCore and FPGA is based on the AXI bus . Similarly, the communication between the FPGA and the main memory is also based on the AXI bus. In other words, the ARM MPCore can access the FPGA through the AXI bus, while the FPGA can also use the AXI bus to access the main memory.

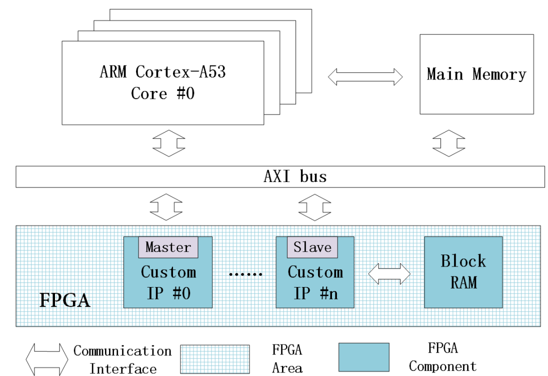

Fig. 1: Simplified Hardware architecture of Zynq UltraScale+ MPSoC (ZU9EG).

On the other hand, the FPGA contains two most important types of components, namely custom IP and block RAM. Block RAM is the storage space owned by the FPGA itself. These block RAMs can be configured to be visible or invisible to the AXI bus. When the block RAMs are configured to be invisible to AXI bus, they can only be accessed by the custom IP of the FPGA. We design SnapMem based on this feature of block RAM. The custom IP represents the hardware programming capability of the FPGA, which is the hardware module designed by the hardware engineer. When the custom

IP needs to communicate with the ARM MPCore or the main memory, an AXI interface needs to be configured for it. AXI interface [43] can be divided into two categories in terms of function, namely master interface and slave interface. As the name implies, the master interface is the interface that can actively access the main memory according to the physical address. Correspondingly, the slave interface means that the custom IP has its own physical address and can be accessed by ARM MPCore. In the design of SnapMem, both AXI master and slave interfaces are required.

B. Cache-Related Attacks

Cache-related attacks can be divided into three categories, namely low resolution cache attacks, high resolution cache attacks and transient execution Attacks. Low resolution cache attack is a cache attack launched by early security researchers using statistical methods [2], [5], [8], [9], [44], [45]. Because this type of attack has a lot of noise and the attack efficiency is very low, it is called low resolution cache attack. In the second decade of the 21st century, cache attacks based on cache flush instructions or operations began to emerge, typically such as Evict+Reload [11], Flush+Reload [10], and Flush+Flush [13]. Lipp et al. [14] successfully implemented the Evict+Reload, Flush+Reload and Flush+Flush attacks on ARMv8-A. Beginning in 2018, transient execution attacks have entered the field of vision of researchers, including Spectre-BTB [16], Spectre-PHT [16], [17], Spectre-STL [18], Meltdown [15], Meltdown-GP [23], MeltdownPrime and SpectrePrime [25], etc. This type of attack exploits the transient execution vulnerability of the CPU and steals sensitive information through the cache side channel. Therefore, this type of attack is also an important category of cache-related attacks.

C. Defenses and Limitations

Hardware Defenses: The most effective defense solution for Meltdown, Spectre and their variants is to modify the hardware architecture of modern processors. In the past three years, Many hardware defense schemes have been proposed, such as SafeSpec [26], Conditional Speculation [28], Spectre-Guard [30], ConTExT [27], InvisiSpec [46], STT [47], Reuse-trap [29], SpecCFI [31], MuonTrap [32] and GhostMinion [33]. Most of these schemes are effective against Meltdown and Spectre variants and have little performance overhead. However, modifications to the processor hardware architecture can only be implemented on next-generation products. These defenses cannot be deployed on existing devices.

Software Runtime Defenses: Before the discovery of Meltdown and Spectre, Researchers [34], [35], [48] often defend against cache-related attacks by monitoring cache activity in real time. But these runtime defenses are only effective for Prime+Probe or encryption-targeted Flush+Reload. They are incapable of defending against Meltdown, Spectre and their variants. In the Linux kernel of ARMv8-A, there are three runtime defenses that can effectively defend against Meltdown and Spectre variants, which are KPTI (kernel page table isolation) [39], Spectre-BTB mitigation [37] and Spectre-STL mitigation [38] respectively. However, Each of these defenses targets only one Meltdown or Spectre variant. They cannot cover all Meltdown or Spectre variants. Moreover, a certain performance overhead is also brought by each of the three defenses. In addition, there are two software runtime defenses [49], [50] that focus on designing more secure APIs to resist cache-related attacks. But both of the two researches are only effective for cache-related attacks using the flush instructions.

Static Code Fixing: Static code analysis/fixing defenses [40], [51], [52] are effective for defending against or detecting cache-related attacks. However, fixing static code imposes a large performance overhead on the runtime system [40].

Software/Hardware Collaborative Defenses: Two encryption implementations [53], [54] based on software/hardware co-design can resist cache timing attack, which is an old and low resolution cache-related attack. The two defenses are powerless against the new high resolution cache-related attacks. There are two software/hardware collaborative defenses [55], [56] that can resist the latest high resolution cache-related attacks and Spectre/Meltdown variants. However, these two schemes are based on the detection of flush instructions and cannot defend against cache-related attacks that do not require flush instructions, such as Prime+Probe, Evict+Reload, MeltdowmPrime and SpectrePrime.

III. Threat Model

A. Our Threat Model

Our assumptions of the attacker are as follows. First, the attacker can execute her arbitrary code on the same machine with the victim process, but without root privileges. Therefore, the attacker can launch all cache-related attacks, such as Prime+Probe, Evict+Reload, Flush+Reload, Flush+Flush, Meltdown, Spectre and so on. Since the attacker does not have root privileges, she cannot access software or hardware modules of SnapMem. Second, the attacker knows the address layout of the victim process or kernel. In fact, it is not very difficult to obtain the address layout of the victim process or kernel. In many attack scenarios, an attacker can legally obtain a copy of the victim's process or kernel binary. Using a binary copy of the victim's process or kernel, the attacker can load and fully reproduce the address layout on his or her host. The only difference between the address layout of these copies running on the attacker's host and the victim process or kernel is the base address. For the victim process, the attacker only needs to exploit vulnerabilities such as memory reading to easily obtain its base address. As for the base address of the victim kernel, it often follows certain operating system rules. For example, in ARM64-based OS, the kernel base address is usually 0xFFFF000000000000, while in x86_64-based OS, the kernel base address is usually 0xFFFF800000000000. Based on the obtained address layout, the attacker can infer the virtual and physical addresses of sensitive data in the victim process or kernel space. This assumption enables attackers to clean the targeted cache lines to launch high resolution cache-related attacks.

In summary, the attacker can implement all cross-domain cache-related attacks to steal sensitive data throughout the main memory. However, the attacker cannot access SnapMem's modules and thus cannot affect SnapMem's data and

control flow. Therefore, the attacker has one and only one way to access SnapMem, which is to call SnapMem's APIs.

B. Threat Model Comparison

In order to explain SnapMem's threat model more clearly and intuitively, we compare SnapMem with existing defense solutions. According to the introduction in Section II-C, defense solutions can be divided into four categories, namely hardware defenses, software runtime defenses, static code fixing and software/hardware collaborative defenses. Among them, the threat models of the first two categories are relatively clear, while the threat models of the latter two categories are relatively vague. Therefore, we focus on comparing SnapMem with hardware defenses and software runtime defenses.

The first category is hardware defense solutions that modify the hardware architecture, such as Conditional Speculation [28], MuonTrap [32], SPECCFI [31], SpectreGuard [30], InvisiSpec [46] and STT [47]. The threat models for this type of hardware defense solution are almost identical and can be summarized into three assumptions. The first assumption of the hardware defense solutions is the same as SnapMem, which is that the attacker can run code on the same machine as the victim process. The second assumption of the hardware defense solutions is quite different from SnapMem. SnapMem assumes that attackers can launch all cache-related attacks, including transient execution attacks (Meltdown, Spectre and their variants), as well as traditional cache-related attacks (Prime+Probe, Evict+Reload, Flush+Reload and Flush+Flush). The hardware defense solutions assume that attackers can only launch transient execution attacks. The third assumption of the hardware defense solutions is the same as SnapMem, which also requires the attacker to obtain the address layout of the victim process. As can be seen from the above comparisons, the threat model of the hardware defense solutions make smaller assumptions about the attacker's capabilities than SnapMem's.

The second category is software defense solutions, such as KPTI [39], HomeAlone [34] and CloudRadar [35]. The threat model assumptions of these three software defense solutions are very similar to SnapMem. However, when these three software defense solutions were proposed, transient execution attacks had not yet been discovered. Therefore, their threat models only assume traditional cache-related attacks and do not include transient execution attacks. In other words, these three software defense solutions have lower assumptions about attacker capabilities than SnapMem.

Overall, SnapMem's threat model makes higher assumptions about attacker capabilities than existing defense solutions. In other words, SnapMem's defense capabilities are more powerful than existing defense solutions.

IV. DESIGN

In this section, we first introduce the entire design of SnapMem. Then, we will provide details of the software and hardware modules of SnapMem.

A. Overview of SnapMem

SnapMem is a more secure memory mechanism for sensitive data. Its core idea is to hide sensitive data where the CPU cannot access it unless the SnapMem's APIs are called. SnapMem's APIs have a process-based authentication mechanism. In other words each process can create its own SnapMem, but can only access the SnapMem created by itself. Similarly, a malicious process can create its own SnapMem, but cannot access the SnapMem of other processes. Even if the malicious process can read the entire main memory using cache-related attack, it still cannot steal sensitive data in SnapMem. Because the sensitive data is stored in SnapMem's hardware memory, not in the main memory. And the hardware memory of SnapMem can only be accessed by the hardware module of SnapMem, not by the CPU.

Figure 2 shows the entire hardware and software architecture of SnapMem. From Figure 2, we can see that the entire software and hardware system is divided into three layers, namely user space, kernel space and hardware layer. SnapMem has components in all three layers. In user space, SnapMem has a total of 4 API components, which are SnapMem_alloc(), SnapMem_read(), SnapMem_write() and SnapMem_free() respectively. As the name implies, the function of SnapMem_alloc() is to allocate hardware memory of SnapMem. The functions of SnapMem_read() and SnapMem_write() are to read and write SnapMem's hardware memory respectively. SnapMem_free() completes the release of SnapMem's hardware memory.

In kernel space, the design of SnapMem mainly includes SnapMem software module and the modifications to kernel functions (_do_fork() and do_exit()). SnapMem software module implements the functions of 4 SnapMem's APIs in kernel space. The work of process identity authentication is done by the submodule of *Process ID Checker*. In the hardware layer, SnapMem has two components, SnapMem hardware module and SnapMem memory. SnapMem hardware module completes the functions of communicating with software and transferring data from hardware to the main memory. The submodule of *Controller* is mainly responsible for the function of communicating with software. The work of data movement is completed by another submodule *Data Mover*. The function of SnapMem memory is to store sensitive data on hardware.

To reduce the overhead under certain conditions, we design two versions of SnapMem, named SnapMem-strict and SnapMem-light. The strict version of SnapMem will actively clear sensitive data in the main memory after each read and write. The light version of SnapMem does not actively clear the main memory after each read and write. SnapMem-light will passively clear the main memory only when the legitimate process actively calls the two APIs SnapMem_light_open() and SnapMem_light_close() to close the SnapMem.

The data and control flow of SnapMem-strict are shown in Figure 3. For simplicity and intuition, Figure 3 focuses on reading and writing to SnapMem and omits SnapMem_alloc() and SnapMem_free(). From Figure 3 we can see that there are 8 steps for reading and 6 steps for writing to SnapMem. Below, we introduce the 8 steps of SnapMem_read() and 6

Fig. 2: Hardware and software architecture of SnapMem

steps of SnapMem_write() in detail respectively.

In summary, SnapMem_read() (strict) involves the following 8 steps.

- ¬ Process ID Checker checks if the caller is legal.
- - Software module communicates with Controller to tell the hardware what memory size and offset to access.
- ® Controller passes the size and offset information to Data Mover, and starts the read function of Data Mover.
- ¯ Data Mover reads out the data of the corresponding address in the hardware memory one by one.
- ° Data Mover sends the read data into the main memory.
- ± Software module transfers data from kernel to user space.
- ² Data Mover reads out the data in the main memory and clears the main memory one by one.
- ³ Data Mover sends the read data into the hardware memory.

Correspondingly, SnapMem_write() (strict) has similar 6 steps as shown below.

1) The identity of the caller is checked by Process ID Checker.
2) Software module transfers data from user to kernel space.
3) Software module passes the information of memory size and offset to Controller.
4) Controller tells Data Mover the memory size and offset.
5) Data Mover reads out the data of the corresponding address in the main memory, and clears the main memory.
6) Data Mover sends the read data into the hardware memory.

In the SnapMem-light design, SnapMem_read() and SnapMem_write() are much simpler than SnapMem-strict. In SnapMem-light, SnapMem_read() only needs to run steps ¬ and ±, while SnapMem_write() only needs to complete steps 1) and 2). All other steps are completed by the two APIs SnapMem_light_open() and SnapMem_light_close(). We will describe the differences between the two versions in more details in Section IV-B.

B. Software of SnapMem

As described in Section IV-A, the main function of SnapMem software module is to complete the operations of the four APIs in kernel space. Below, we describe the functional design of these four APIs in detail. Additionally, we will introduce modifications to the original kernel functions to support SnapMem. For ease of introduction, we list all parameters and signals that the software module of SnapMem needs to use in Table I.

TABLE I: The explanations of all the parameters and variables used in software of SnapMem.

Parameter/Signal	Explanation
SnapMem_offset	The offset of the data address to be accessed in SnapMem.
SnapMem_size	The size of the data to be accessed in SnapMem.
SnapMem_data	Data written to or read from SnapMem.
SnapMem_state	Indicates whether SnapMem's status is strict or light.
SnapMem_ID	The SnapMem of each process has its own unique ID.
SnapMem_in	Trigger to transfer data from main memory to SnapMem memory.
SnapMem_out	Trigger to transfer data from SnapMem memory to main memory.

SnapMem_alloc(): SnapMem_alloc() mainly completes three operations. First, query whether the current process has created a SnapMem. Exits with an error if the current process has already created a SnapMem. Second, allocate SnapMem for the current process and store the process ID. Third, assign an SnapMem_ID to the current process.

SnapMem_read(): The operation steps of SnapMem_read() in kernel space are shown in Algorithm 1. A detailed description of all parameters and variables can be found in Table I. From Algorithm 1 we can see that if the process owns a SnapMem, it will get its own unique SnapMem_ID. The SnapMem_ID determines which SnapMem memory it can

Fig. 3: Data and control flow of SnapMem-strict.

Algorithm 1: SnapMem_read() of SnapMem

Input: Offset of SnapMem SnapMem_offset;
Size of SnapMem SnapMem_size;
Output: Data read from SnapMem SnapMem_data;

```
1  if The current process has a SnapMem then
2  |  Get its own SnapMem_ID;
3  |  if SnapMem_state is SnapMem-strict then
4  |  |  Flush and Invalidate all cache lines of the main
   |  |    memory corresponding to SnapMem;
5  |  |  Pass the SnapMem_ID to Controller;
6  |  |  Pass SnapMem_offset and SnapMem_size to
   |  |    Controller;
7  |  |  Send a SnapMem_out trigger to Controller;
8  |  |  Invalidate all cache lines of the main memory
   |  |    corresponding to SnapMem;
9  |  |  Waiting for the Data Mover to complete its
   |  |    work;
10 |  |  Transfer SnapMem_data in the main memory
   |  |    from kernel space to user space;
11 |  |  Flush and Invalidate all cache lines of the main
   |  |    memory corresponding to SnapMem;
12 |  |  Pass the SnapMem_ID to Controller;
13 |  |  Pass SnapMem_offset and SnapMem_size to
   |  |    Controller;
14 |  |  Send a SnapMem_in trigger to Controller;
15 |  |  Invalidate all cache lines of the main memory
   |  |    corresponding to SnapMem;
16 |  |  Waiting for the Data Mover to complete its
   |  |    work;
17 |  else
18 |  |  Transfer SnapMem_data in the main memory
   |  |    from kernel space to user space;
19 else
20 |  Return an error to user space;
```

access. The related operation of this SnapMem_ID completes the authentication mechanism of the process. After getting SnapMem_ID, you need to check the status of SnapMem. This status determines whether SnapMem is a strict version or a light version. Starting from the third line of Algorithm 1, we can clearly see the difference between SnapMem-strict and SnapMem-light. The read operation of SnapMem-light omits the steps of controlling the hardware, so its steps are very simple. From the fourth line of Algorithm 1 we know that cache lines need to be flushed and invalidated before passing parameters to Controller. The purpose of these steps is to ensure that all data can be flushed from the cache back to the main memory. This ensures that the hardware transmits the correct data in the main memory.

After all parameters are passed to the hardware, we need to start the hardware to transfer the data from SnapMem memory into the main memory, as shown in line 7 of Algorithm 1. Subsequently, the cache line will be invalidated again in line 8. This step is to ensure that CPU access to SnapMem_data bypasses the cache and directly accesses the main memory. When the hardware completes the data transfer work, SnapMem_data will be moved from kernel to user space in line 10. Then, operations related to the cache lines and controlling the hardware are re-executed. The difference is that the data transfer direction of the hardware is reversed, from the main memory to SnapMem memory, as shown in lines 11 to 16. This reverse data movement is to re-hide sensitive data in hardware memory.

SnapMem_write(): Algorithm 2 shows the operation steps of SnapMem_write() in kernel space. As can be seen from Algorithm 2, the operation steps of SnapMem_write() are very similar to SnapMem_read(). SnapMem_ID and SnapMem_state related operations are the same as SnapMem_read(), as shown in lines 1 to 3. In addition, SnapMem_write() does not require cache line and hardware related operations before transferring data from user to kernel space.

Algorithm 2: SnapMem_write() of SnapMem

```
   Input: Offset of SnapMem SnapMem_offset;
   Size of SnapMem SnapMem_size;
   Output: Data written to SnapMem SnapMem_data;
 1 if The current process has a SnapMem then
 2     Get its own SnapMem_ID;
 3     if SnapMem_state is SnapMem-strict then
 4         Transfer SnapMem_data in the main memory
            from user space to kernel space;
 5         Flush and Invalidate all cache lines of the main
            memory corresponding to SnapMem;
 6         Pass the SnapMem_ID to Controller;
 7         Pass SnapMem_offset and SnapMem_size to
            Controller;
 8         Send a SnapMem_in trigger to Controller;
 9         Invalidate all cache lines of the main memory
            corresponding to SnapMem;
10         Waiting for the Data Mover to complete its
            work;
11     else
12         Transfer SnapMem_data in the main memory
            from user space to kernel space;
13 else
14     Return an error to user space;
```

Lines 5 to 10 show the steps for transferring SnapMem_data from the main memory to the hardware memory, which is the same as lines 11 to 16 of SnapMem_read(). In general, SnapMem_write() is much simpler than SnapMem_read(), because it only runs the operations related to the cache line and hardware control once.

SnapMem_free(): SnapMem_free() mainly runs two operations. First, clear the stored current process ID to zero. Second, clear the SnapMem_ID value assigned to the current process to zero.

Modifications to Kernel: SnapMem's modifications to the kernel are mainly for two kernel functions: _do_fork() and do_exit(). The two functions are run when the process is created and terminated, respectively. We modify them mainly to prevent the memory leak of SnapMem. As we all know, unexpected termination of a process occurs frequently in a running system. It will cause the allocated SnapMem to not be released. On the other hand, the release of SnapMem requires a call to SnapMem_free(). If the programmer accidentally forgets to call SnapMem_free() in the end of program, SnapMem cannot be released too. The occurrence of the two situations will lead to memory leaks of SnapMem. On ARM-FPGA SoC, SnapMem is an invaluable resource. We cannot tolerate the occurrence of these two SnapMem memory leaks. Therefore, we modified _do_fork() and do_exit() so that the process checks for memory leaks of SnapMem on both creation and termination. The operation they add is to check whether the current process has its own unreleased SnapMem. If there is, release the SnapMem, clear the stored process ID and SnapMem_ID to zero.

C. Hardware of SnapMem

As can be seen from Figures 2 and 3, SnapMem hardware module consists of two submodules, namely *Controller* and *Data Mover*. Below, we describe the operation steps of these two submodules in detail. The introduction of SnapMem memory is omitted here. Because SnapMem memory is created based on Zynq's block RAM, it uses Zynq's free commercially available mature interface IPs. In other words, SnapMem memory does not have any innovation in functionality. Therefore, the functional details of SnapMem memory will not be introduced here.

Controller: The function of *Controller* is mainly to complete the control of *Data Mover* and the conversion of addresses or signals. *Controller* has a total of 4 operation steps as follows.

1. Receive SnapMem_ID, SnapMem_offset, SnapMem_size and the trigger signal sent by software.
2. Based on SnapMem_ID, SnapMem_offset and SnapMem_size, calculate the address range of SnapMem_data.
3. Send the address range of SnapMem_data to *Data Mover*.
4. Send the trigger signal to *Data Mover* to start its data transfer function.

Fig. 4: State machine of Data Mover in SnapMem.

Data Mover: In SnapMem hardware module, *Data Mover* is the core component. Figure 4 shows the state machine of *Data Mover*. From Figure 4, we can know that *Data Mover* has a total of 5 states, namely *Idle*, *Controller_Clear*, *Read*, *Memory_clear* and *Write*. Table II lists all registers and variables that *Data Mover* needs to use. Below, we describe the operation steps in the 5 states of *Data Mover* in detail.

TABLE II: The explanations of all the Registers and variables used in *Data Mover*.

Register/Signal	Explanation
SourceStartAddr	The start address of the source data.
SourceEndAddr	The end address of the source data.
DestinStartAddr	The starting address of the destination data.
DestinEndAddr	The end address of destination data.
DataMovTrig	The trigger register for *Data Mover*.
WrAddrReg	The register that stores the destination write address.
WrDataReg	The register that stores the data to be written to destination address.
RdAddrReg	The register that stores the source address of the read data.
RdDataReg	The register to store read data.

- Idle: This is the initial state of Data Mover. In the Idle state, the main function of Data Mover is to wait for the input of signals and parameters, including SourceStartAddr, SourceEndAddr, DestinStartAddr, DestinEndAddr and DataMovTrig.
- Controller_Clear: When DataMovTrig is not equal to 0, Data Mover will enter Controller_Clear state from the Idle state. In this state, Data Mover mainly completes one operation, that is, clearing DataMovTrig register of Controller to zero. If this DataMovTrig register is not cleared, Data Mover will be triggered again to start after completing the data transfer and returning to the Idle state.
- Read: This state has two functions. The first function of Read state is to read the data at the source address into RdDataReg of Data Mover. Then, Data Mover puts the read data into WrDataReg and waits to be written to the destination address.
- Memory_Clear: There are two operations in this state. The first operation of this state is to clear the data of the source address of the read data. Then, RdAddrReg register will be incremented by 4.
- Write: In this state, Data Mover first writes the data in WrDataReg to the destination address. Then, Data Mover will judge whether WrAddrReg is equal to DestinEndAddr. If WrAddrReg is equal to DestinEndAddr, Data Mover goes back to the Idle state. If WrAddrReg is not equal to DestinEndAddr, WrAddrReg is incremented by 4.

V. Implementation

Section IV focuses on the design principle of SnapMem. In this section, we will introduce the implementation of SnapMem in detail. Like Section IV, we also describe the implementation from both software and hardware aspects.

A. Software of SnapMem

SnapMem_alloc(): We use the ioctl() system call to pass a parameter to the SnapMem software module in the kernel. This parameter instructs the kernel to allocate SnapMem memory. We define a global array SnapMem_PID_list in the kernel. Each element of SnapMem_PID_list is used to hold the ID of the process that owns SnapMem. The index of the element is its SnapMem_ID. By iterating over the elements of SnapMem_PID_list, the first empty element can be found. Then, the ID of the current process is stored in this element.

SnapMem_read(): SnapMem_read() uses the read() system call to enter the kernel and passes two parameters, SnapMem_offset and SnapMem_size. The SnapMem_ID can be obtained by traversing SnapMem_PID_list. We use the two instructions DC CIVAC and DC IVAC to flush and invalidate the cache line. Then, we use iowrite32() to write the three parameters SnapMem_ID, SnapMem_offset, SnapMem_size and the trigger signal to Controller. The kernel function copy_to_user() is utilized by us to move data from kernel space to user space.

SnapMem_write(): The implementations of SnapMem_write() and SnapMem_read() are similar, with only two differences. First, SnapMem_write() enters the kernel using the write() system call. Second, moving data from user to kernel space is done based on copy_from_user().

SnapMem_free(): The implementation of SnapMem_free() is also based on the ioctl() system call. After the kernel traverses the array SnapMem_PID_list, it finds the element that stores the current process ID and clears it to zero.

Modifications to Kernel: The operation steps for adding _do_fork() and do_exit() are the same as SnapMem_free(). The two modified kernel functions traverse SnapMem_PID_list and clear the element storing the current PID.

B. Hardware of SnapMem

We wrote the hardware code for SnapMem using Verilog HDL language. Moreover, we synthesized and implemented the hardware of SnapMem in Vivado Design Suite.

Controller: We configured an AXI slave interface for Controller. Through AXI SmartConnect [57] of Xilinx, we connect the AXI slave interface of Controller to M_AXI_HPM0_FPD interface [43] of the ARM MPCore processor. Therefore, the processor can access Controller through the physical address of the AXI slave interface.

Data Mover: Because Data Mover needs to actively access the main memory and the hardware memory, it is configured with an AXI master interface. Similar to Controller, Data Mover is also based on AXI SmartConnect and is connected to S_AXI_HP0_FPD [43] of the ARM MPCore processor.

VI. Evaluation

In this section, we detail the security, performance and hardware evaluation of SnapMem. First, our evaluation environment is introduced in Section VI-A. Then we describe the security evaluation results in Section VI-C. In Section VI-D, we show the performance evaluation results. Finally, hardware overhead is provided in Section VI-E.

A. Evaluation Environment

Our experimental platform is Xilinx ZCU102 Evaluation Board [58]. It has one ZU9EG SoC with a total of 4 ARM Cortex-A53 cores. The platform also has 4GB of DDR4 memory and 29GB of hard drive storage. We implemented SnapMem and other defense solutions based on Linux kernel 5.4.0 running in Ubuntu 16.04.6 LTS on ZCU102 platform. The cross compiler we utilize is aarch64-linux-gnu- with the gcc version of 10.1.0.

B. Evaluation Design

The evaluation experiments are divided into security analysis, performance analysis and hardware overhead. In the security analysis, all attacks are running on the real ZU9EG SoC platform. However, due to the weak transient execution capability of ZU9EG, transient execution attacks are simulated. For a detailed introduction to simulated attacks, please refer to Section VIII. Performance analysis is divided into

three aspects: API, encryption application and system performance overhead. API experiments mainly focus on the time overhead of API calls. Encryption applications use SnapMem to implement a more secure AES implementation. The system performance experiments compare the performance overhead of SnapMem, KPTI, Spectre-BTB mitigation and Spectre-STL mitigation. running on the ZU9EG SoC platform. One thing that needs to be emphasized is that in the API comparison experiment, the overhead test of reading and writing a small amount of data was also included. The specific results can be found in Table IV. In this test, the hardware defense solutions (STT and InvisiSpec) was running in the Gem5 emulator, while the software defense solution and SnapMem were running on ZU9EG platform. The reason why Gem5 is not used to simulate the SnapMem solution is because the block RAM used by SnapMem has Xilinx's unique interface IPs and timing logic. Using Gem5 to simulate SnapMem will produce great distortion, and ultimately lose the authenticity of the overhead comparison results.

C. Security Analysis

We have tested all cache-related attacks that can be implemented on this ARM-FPGA SoC. Since the branch prediction and out-of-order execution capabilities of ARM Cortex-A53 core are not strong enough, Meltdown, Spectre and their variant attacks cannot actually run on the platform. Therefore, we simulated these types of variant attacks using /dev/mem of the Linux file system. Experimental results in Table III prove that SnapMem can effectively defend against all cache-related attacks, including Prime+Probe, Evict+Reload, Flush+Reload, Flush+Flush, Spectre-PHT, Spectre-BTB, Spectre-STL, SpectrePrime, Meltdown, Meltdown-GP and MeltdownPrime.

TABLE III: Security Evaluation of SnapMem

Attack Classification	Defense Capability
Prime+Probe	X
Evict+Reload	X
Flush+Reload	X
Flush+Flush	X
Spectre-PHT	X
Spectre-BTB	X
Spectre-STL	X
SpectrePrime	X
Meltdown	X
Meltdown-GP	X
MeltdownPrime	X

D. Performance Analysis

APIs of SnapMem: In the performance evaluation experiments, we first test the latency of 4 SnapMem APIs. To make the results look more intuitive, we compared the different SnapMem APIs with the original Linux API of similar functionality. Figure 5 shows the comparison results of SnapMem_malloc() and malloc(). As can be seen from Figure 5, the call delay of SnapMem_malloc is much larger than malloc(), which is about 10 times. One of the reasons for such a big difference is that the call delay of malloc() itself is very small. The second reason is because the process authentication mechanism of SnapMem_alloc() takes much more time. From the Figure 5, we can also know that the call delay of the two APIs has nothing to do with the allocated memory size.

Fig. 5: Comparison of SnapMem_malloc() and malloc().

Fig. 6: Comparison of SnapMem's access APIs and memcpy().

Figure 6 shows the latency comparison between SnapMem's access APIs and memcpy(). This result includes both SnapMem-strict and SnapMem-light for read and write operations. The ordinate in Figure 6 is exponentially growing. As can be seen from Figure 6, the latency of SnapMem-light's read and write functions and memcpy() are in an order of magnitude. Also, as the data size increases, the access latency of SnapMem-light gets closer to memcpy(). This is enough to prove that the access overhead of SnapMem-light is indeed small. In contrast, the access overhead of SnapMem-strict is much greater than the previous two. This is mainly because each read and write operation of SnapMem-strict is accompanied by operations and waits on hardware and cache lines. Therefore, the SnapMem-light is more suitable for low security but high performance requirements, while SnapMem-strict is the opposite.

However, we can also see from Figure 6 that SnapMem is not suitable for reading and writing small amounts of data. When reading and writing 128-byte data, whether it is SnapMem-strict or SnapMem-light, the performance overhead of reading and writing is very large. The results in Table IV more clearly show the comparison of read and write overhead for a small amount of data between different defense solutions. Among hardware defense solutions, STT and InvisiSpec are two open source projects that can be simulated and evaluated on Gem5 [59]. We used the Gem5 simulator to simulate the STT and InvisiSpec solutions respectively, and tested the time overhead of memcpy(). On the other hand, we also tested the time overhead of memcpy() of three software defense solutions

TABLE IV: Reading and writing overhead for small amounts of data (128 bytes). The overheads of hardware solutions (STT and InvisiSpec) are compared with the original CPU hardware architecture, and the overheads of software solutions (KPTI, Spectre-BTB mitigation and Spectre-STL mitigation) are compared with the original Linux OS. The overhead of SnapMem is also compared to the memcpy() function in the original Linux OS.

Solution	Function	Overhead
STT	memcpy()	27%
InvisiSpec	memcpy()	35%
KPTI	memcpy()	5%
Spectre-BTB mitigation	memcpy()	3%
Spectre-STL mitigation	memcpy()	2%
SnapMem-light	SnapMem_light_read()	1120%
	SnapMem_light_write()	900%
SnapMem-strict	SnapMem_strict_read()	4135540%
	SnapMem_strict_write()	8284700%

KPTI, Spectre-BTB mitigation and Spectre-STL mitigation. From Table IV, we can see that in the case of small data volume, the overhead of SnapMem is far higher than that of other solutions. Therefore, the current SnapMem design is not suitable for read and write operations of small data volumes. In addition, it can be seen from Table IV that the cost of hardware defense solutions is much greater than that of software defense solutions. This is because the hardware defense solution modifies the cache hierarchy or predictor, which greatly affects the memory read and write speed. Correspondingly, the modification of the kernel source code by the software defense solution has very little impact on the memory read and write speed.

Fig. 7: Comparison of SnapMem_free() and free().

Figure 7 shows the call delay comparison results of SnapMem_free() and free(). This result is similar to that of Figure 5. The big delay difference between SnapMem_free and free() is also because of SnapMem_free's process authentication mechanism.

AES Encryption: In order to see the overhead of SnapMem in practical applications, we designed the T-table encryption implementation of AES based on the two versions of SnapMem. Figure 8 shows the encryption time comparison between the original T-table AES, SnapMem-strict-based AES and SnapMem-light-based AES implementations. As can be seen from Figure 8, when the amount of data is relatively small, the overheads of the two SnapMem-based AES implementations are much larger than the original version of AES. But as the amount of data continues to increase, the time of the three is getting closer and closer. This result shows that SnapMem has good performance overhead in the case of large data volumes, but poor performance in the case of small data volumes.

Fig. 8: Comparison of AES encryption time.

UnixBench [60]: Researchers design UnixBench to provide a basic indicator of the performance of a Unix-like system. We perform UnixBench under 5 system settings to test the performance overhead of critical operations on Linux system. Figure 9 shows the comparative evaluation results of UnixBench. As can be seen from Figure 9, the test Context Switching increase significantly on the SnapMem enabled Linux system. This is mainly because Context Switching creates and terminates a large number of processes, while SnapMem has additional operations to release SnapMem memory in the creation and termination of processes.

Overall, KPTI enabled Linux has the largest total overhead on UnixBench, which is as high as 1.5%. The total overheads of the Spectre-BTB mitigation and the Spectre-STL mitigation are 0.98% and 0.6%, respectively. Correspondingly, SnapMem's UnixBench has a total overhead of just 0.07%. This result indicates that SnapMem has much less impact on the critical operations of Linux system than the other three defenses.

SPEC_CPU 2017 [61]: SPEC_CPU 2017 suites provide a comparative measure of compute-intensive performance using workloads developed from real user applications. So we utilize SPEC_CPU 2017 to compare the impact of SnapMem and other defenses on user application performance. We evaluate all of SPEC_CPU 2017 INT and FP applications under five system settings. These 5 system settings are the same as UnixBench tests. Figure 10 shows the evaluation results of SPECrate2017 benchmark. As shown in Figure 10, SnapMem has a high overhead in namd and xalancbmk test. Because both namd and xalancbmk create and terminate processes frequently.

Overall, in the SPEC_CPU 2017 tests, the KPTI mitigation has the highest total overhead at 0.14%. Followed Spectre-STL mitigation, the total overhead is 0.08%. Both SnapMem and Spectre-BTB mitigation have a total overhead of 0.04%. This result indicates that SnapMem and Spectre-BTB mitigation has less performance impact on user applications than the other two defenses.

Fig. 9: Evaluation results of UnixBench.

Fig. 10: SPEC2017 benchmark results.

E. Hardware Overhead

Table V provides the hardware implementation overhead of SnapMem. As can be seen from Table V, in the hardware design of the entire SnapMem, the AXI interface occupies the most hardware resources, which are 87% of all LUTs and 89% of all registers, respectively. Relatively speaking, Data Mover and Controller occupy much less hardware resources than AXI interface. Data Mover uses 5.8% of LUTs and 1.3% of registers, while Controller utilizes 2.9% of LUTs and 6.1% of registers. On the other hand, SnapMem memory takes up all the block RAM. Moreover, SnapMem memory also uses 3.9% of LUTs and 3.8% of registers due to the need for a communication interface.

TABLE V: Hardware implementation overhead of SnapMem

SnapMem Component	LUT as logic	LUT as Memory	Registers	Block RAM
Data Mover	1025	0	303	0
Controller	515	0	1379	0
AXI Interface	11307	3999	20009	0
SnapMem Memory	685	5	850	360Kb
Reset Module	13	1	34	0
Overall	13545	4005	22575	360Kb

VII. Related Work

In this section, we present several software runtime defenses and software/hardware co-design to resist cache-related attacks similar to SnapMem.

Several effective studies on software runtime defense are proposed to resist Prime+Probe, Flush+Reload and Flush+Flush attacks. HomeAlone [34] can detect Prime+Probe attacks by using the same side channel (L2 cache) as attackers. CloudRadar [35] can detect Prime+Probe and Flush+Reload attacks in real time using hardware performance counters. Secure Collaborative APIs (SCAPI) [50] designed a collaborative mechanism for flush and time APIs to resist Flush+Reload and Flush+Flush in real time.

Some effective software runtime defenses against Spectre, Meltdown and their variants are also proposed by researchers. Google presents Return Trampoline (retpoline) [36] to defend against Spectre-BTB. It replaces indirect branches with push+return instruction sequences to prevent BTB poisoning. kernel Page-Table Isolation (KPTI) [39] can defend against Meltdown because it ensures no valid mapping to kernel space in user space. EPTI [62] was designed to defend against Meltdown attack in cloud. It has less overhead than KPTI and can be applied to unpatched VMs. Additionally, Linux open source community provides two software runtime schemes on ARMv8-A, which are Spectre-BTB mitigation [37] and Spectre-STL mitigation [38] respectively.

There are also some studies that utilize software/hardware co-design to defend against cache-related attacks in real time. AdapTimer [55] is a software/hardware collaborative timer, which can resist flush-based cache-related attacks on ARM-FPGA embedded SoC. SecFlush [56] designs a flush detector in hardware, which cooperates with software to mitigate cache-related attacks.

VIII. DISCUSSION

Simulated Attacks: All experiments of SnapMem are based on Xilinx's ZU9EG platform, which has 4 ARM Cortex-A53 cores. However, due to the weak branch prediction and out-of-order execution capabilities of the ARM Cortex-A53 multi-core processor, we cannot successfully run Meltdown, Spectre-BTB, and Spectre-STL attacks on this platform. Therefore, we can only run simulated experiments for these three attacks. The simulated Meltdown, Spectre-BTB and Spectre-STL attacks utilize /dev/mem of the Linux file system to directly read and write the main memory corresponding to SnapMem. Although they are not real attacks, they have a greater ability to steal sensitive data than real attacks. The failure of these three simulated attacks is enough to prove the security of SnapMem.

Overhead of API: From the performance evaluation results, we can see that SnapMem has a very large overhead when reading and writing a small amount of data. This is because every read and write of SnapMem starts the hardware module regardless of the data size. In future work, we look forward to solving the problem of large overhead with small amounts of data. A feasible solution is to directly make the hardware memory of SnapMem accessible to the CPU when reading and writing small amounts of data. SnapMem's memory is only accessible when SnapMem's read and write APIs are called, and is inaccessible at other times. Here, we temporarily name this solution SnapMem_Opt. The core of the SnapMem_Opt solution is a control register that determines whether SnapMem can be directly accessed by the CPU. This controller is a privileged controller and can only be controlled by SnapMem's API in user space. Since this solution omits the process of moving data from SnapMem memory to main memory, it greatly reduces the reading and writing overhead of a small amount of data. We use Gem5 simulator to design an proof-of-concept version of SnapMem_Opt and conduct a feasibility assessment of the read and write overhead. To be typical without losing generality, the SnapMem_Opt solution does not use dedicated block RAM and interface IPs, but is designed based on a separately isolated physical space in the main memory. Due to time constraints, this proof-of-concept version of SnapMem_Opt did not perform any optimization of hardware and software. Therefore, it is far from optimal performance and full functionality. But it is enough to verify the feasibility of the technical route. Table VI shows the comparison of read and write overhead of small amounts of data between SnapMem_Opt and SnapMem.

TABLE VI: Comparison of reading and writing overhead for small amounts of data (128 bytes) between SnapMem_Opt and SnapMem.

Solution	Function	Overhead
SnapMem-light	SnapMem_light_read()	1120%
	SnapMem_light_write()	900%
SnapMem-strict	SnapMem_strict_read()	4135540%
	SnapMem_strict_write()	8284700%
SnapMem_Opt	SnapMem_Opt_read()	259%
	SnapMem_Opt_write()	231%

As can be seen from Table VI, compared with SnapMem, the overhead of the proof-of-concept version of SnapMem_Opt has been greatly reduced. This experimental result proves that SnapMem_Opt's technical route is feasible. In our future work, we will conduct comprehensive performance optimization and improvements on the proof-of-concept version of SnapMem_Opt. We believe that the overhead of the improved version of SnapMem_Opt will be further reduced.

IX. CONCLUSION

Cache-related attacks have become a huge security threat to ARM-FPGA embedded SoCs. Existing hardware defenses cannot be deployed on existing SoCs, while software runtime defenses have limited defense capabilities and high performance overhead. This paper presents SnapMem, a software/hardware collaborative defense that can be deployed on existing ARM-FPGA platforms. SnapMem is a more secure memory that burns after reading. Any process can only access the SnapMem created by itself. Sensitive data appears briefly in the main memory only when accessed by a legitimate process calling SnapMem's APIs. Based on this mechanism, SnapMem can resist all cache-related attacks. We implemented various cache-related attacks on real ARM-FPGA platform and verified that SnapMem is more secure than other defense schemes. The system performance evaluation results show that the overheads of SnapMem are the lowest among all defense solutions.

REFERENCES

[1] Xilinx, "Xilinx adaptive socs," https://www.xilinx.com/products/silicon-devices/soc.html, 2022.

[2] P. C. Kocher, "Timing attacks on implementations of dif e-hellman, rsa, dss, and other systems," in Advances in Cryptology - CRYPTO '96, 1996, pp. 104–113.

[3] J. Kelsey, B. Schneier, D. A. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in ESORICS 98, 1998, pp. 97–110.

[4] D. J. Bernstein, "Cache-timing attacks on aes," https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in CT-RSA 2006, 2006, pp. 1–20.

[6] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential cache-collision timing attacks on AES with applications to embedded cpus," in CT-RSA 2010, 2010, pp. 235–251.

[7] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in S&P 2011, 2011, pp. 490–505.

[8] M. Weiß, B. Heinz, and F. Stumpf, "A cache timing attack on AES in virtualization environments," in FC 2012, 2012, pp. 314–328.

[9] R. Spreitzer and T. Plos, "On the applicability of time-driven cache attacks on mobile devices," in Network and System Security - 7th International Conference, NSS 2013, 2013, pp. 656–662.

[10] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 719–732.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in USENIX Security 15., 2015, pp. 897–912.

[12] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "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, 2015, pp. 1406–1418.

[13] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ ush: A fast and stealthy cache attack," in DIMVA 2016, 2016, pp. 279–299.

[14] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in USENIX Security 16, 2016, pp. 549–564.

[15] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in USENIX Security 18, 2018.

[16] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in S&P'19), 2019.

[17] V. Kiriansky and C. A. Waldspurger, "Speculative buffer over ows: Attacks and defenses," CoRR, vol. abs/1807.03757, 2018. [Online]. Available: http://arxiv.org/abs/1807.03757

[18] J. Horn, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[19] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in 12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018, C. Rossow and Y. Younan, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[20] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[21] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[22] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," 2018. [Online]. Available: $$Uhttps://lirias.kuleuven.be/retrieve/515917$$Dforeshadow-ng.pdf[freelyavailable]

[23] ARM, "Vulnerability of speculative processors to cache timing side-channel mechanism," https://developer.arm.com/support/security-updates/speculative-processor-vulnerability, 2021.

[24] J. Stecklina and T. Prescher, "Lazyfp: Leaking FPU register state using microarchitectural side-channels," CoRR, vol. abs/1806.07480, 2018. [Online]. Available: http://arxiv.org/abs/1806.07480

[25] C. Trippel, D. Lustig, and M. Martonosi, "Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," CoRR, vol. abs/1802.03802, 2018. [Online]. Available: http://arxiv.org/abs/1802.03802

[26] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. ACM, 2019, p. 60. [Online]. Available: https://doi.org/10.1145/3316781.3317903

[27] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," in 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/

[28] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019. IEEE, 2019, pp. 264–276. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00043

[29] H. Fang, M. Doroslovacki, and G. Venkataramani, "Reuse-trap: Re-purposing cache reuse distance to defend against side channel leakage," in 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020. IEEE, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/DAC18072.2020.9218725

[30] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An ef cient data-centric defense mechanism against spectre attacks," in Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. ACM, 2019, p. 61. [Online]. Available: https://doi.org/10.1145/3316781.3317914

[31] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Specc : Mitigating spectre attacks using CFI informed speculation," in 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. IEEE, 2020, pp. 39–53. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00033

[32] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in 47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020. IEEE, 2020, pp. 132–144. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00022

[33] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021. ACM, 2021, pp. 592–606. [Online]. Available: https://doi.org/10.1145/3466752.3480074

[34] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in 32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA. IEEE Computer Society, 2011, pp. 313–328. [Online]. Available: https://doi.org/10.1109/SP.2011.31

[35] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings, ser. Lecture Notes in Computer Science, F. Monrose, M. Dacier, G. Blanc, and J. García-Alfaro, Eds., vol. 9854. Springer, 2016, pp. 118–140. [Online]. Available: https://doi.org/10.1007/978-3-319-45719-2_6

[36] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[37] W. Deacon, "arm64: Add skeleton to harden the branch predictor against aliasing attacks," https://patchwork.kernel.org/project/linux-arm-kernel/patch/4349161f0ed572bbc6bff64bad94aa96d07b27ff.1562908075.git.viresh.kumar@linaro.org/, 2018.

[38] M. Zyngier, "arm64: Run arch_workaround_2 enabling code on all cpus," http://lkml.iu.edu/hypermail/linux/kernel/2010.3/11148.html, 2020.

[39] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in Engineering