2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-9425-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/TrustCom56396.2022.00166

Cache Design Effect on Microarchitecture Security: A Contrast between Xuantie-910 and BOOM

Zhe Zhou*, Xiaoyu Cheng*, Yang Sun*, Fang Jiang*, Fei Tong*†‡, Yuxing Mao§¶ and Ruilin Wang‖
*Southeast University, Nanjing, Jiangsu, China
‡Purple Mountain Laboratories, Nanjing, Jiangsu, China
§School of Electrical Engineering, Chongqing University, Chongqin, China
¶State Key Laboratory of Power Transmission Equipment & System Security and New Technology, Chongqing University, Chongqing, China
‖School of Computing, Newcastle University, United Kingdom
‡*Corresponding Author: ftong@seu.edu.cn*

***Abstract*—Modern processors make use of optimization techniques such as cache and speculation mechanisms to greatly improve performance. But recent research has found that these techniques can also be exploited by attackers to perform powerful side-channel attacks. A large number of powerful cache-based attacks have been replicated and enhanced over Intel X86- and ARM-based architectures, but there is a relative lack of research on RISC-V-based architectures. Xuantie-910 and BOOM are both RISC-V-based processors. So far, cache-side channels in the unprivileged case of Xuantie-910 have not been proven, while cache attacks against BOOM are proliferating. There are two types of caches, including physically-indexed physically-tagged (PIPT) cache (adopted by Xuantie-910) and virtually-indexed physically-tagged (VIPT) cache (adopted by BOOM), corresponding to two different cache addressing forms. VIPT has higher addressing performance than PIPT, since it can directly obtain cache line index from virtual address. In this paper, we study Xuantie-910 and BOOM to explore the impact of cache design on the security of RISC-V-based microarchitecture. Specifically, we compare the impact of their cache addressing forms on precise flushing of cache lines at specified locations, which plays an important role in cache side-channel attacks. Experimental results show that for the VIPT cache in BOOM, the location-specified cache lines can be accurately flushed, and Spectre attack can be successfully carried out by using the cache side-channel. On the other hand, for the PIPT cache in Xuantie-910, it is impossible for attackers to directly and accurately flush the specified location of cache without affecting performance, which hinders the success of cache side-channel attacks. This provides us with an insight that one can adopt a VIPT-based cache with a mechanism similar to PIPT for preventing the accurate access of cache line index, which can not only keep the advantage of high-performance addressing in VIPT but also improve chip security.**

***Index Terms*—Microarchitecture security, Xuantie-910, BOOM, RISC-V, Cache-based side-channel attack**

I. Introduction

RISC-V is a simple and open-source instruction set architecture (ISA), supporting customized extension. After years of technological evolution, RISC-V has become a commercially available ISA. The volume of RISC-V related research has exponentially increased in the past decade. The tremendous momentum of RISC-V adoption in computing platforms is very clear. Various RISC-V devices from small IoT microcontrollers to multi-core high-performance processors have been taped out [1]. As two typical examples, T-head has taped out Xuantie-910 for cloud and edge computing [2], and Berkeley has taped out the third generation of the Berkeley Out-of-Order Machine (BOOMv3), which is an open-source implementation of the RISC-V superscalar out-of-order core [3] .

Recent disclosure of microarchitectural attacks such as Spectre [4] and Meltdown [5] on speculating cores injects an imperative concern into the microarchitectural design space. Architecture designers must now consider security, in addition to power, performance and area, when evaluating new designs. Those instructions which are incorrectly executed by the out-of-order execution and speculative mechanisms in modern processors and whose results are not displayed at the architecture level due to rollback are called transient instructions [4], [5]. Although the execution of transient instructions does not affect the architectural state and thus cannot be observed at the architecture level, the microarchitectural state may change. Transient execution attack is intended to leak sensitive information by converting the change of microarchitecture state to architecture state, which belongs to the side-channel attack.

Cache-based side-channel attacks exploit the difference in access times of cache hits and misses [6]. In recent years, many side-channel attacks have been proposed such as flush+reload [7] and evict+reload [8]. Le et al. [9] implemented a cache-based side channel attack on BOOM, using the indexing of virtual addresses to precisely flush cache. In addition, as cache is one of the most important performance components in modern processor, disabling cache which would cause unacceptable performance degradation is not a practical solution to these attacks. Thus the design of cache is very important for microarchitectural security.

In this paper, we first analyze and compare the different

This work is supported in part by the National Natural Science Foundation of China (No. 61971131), in part by "Zhishan" Scholars Programs of Southeast University, and in part by the scholarship of State Key Laboratory of Power Transmission Equipment & System Security and New Technology (2007DA105127).

2324-9013/22/$31.00 ©2022 IEEE
DOI 10.1109/TrustCom56396.2022.00166

cache designs in Xuantie-910 and BOOMv3, and the impact of these differences on the implementation of Spectre attack which needs to build cache-based side channels by accurately flushing cache. We conclude that the cache design of BOOMv3 allows flushing of the precise location of cache, which is an important step in a successful cache-based side-channel attack. In contrast, Xuantie-910 does not achieve the precise position flushing of cache. Then we implement a cache-based side-channel attack on BOOMv3 and Xuantie-910 through software simulation and/or FPGA prototype verification to prove our analysis, and provide significant cache design insights for improving the security of RISC-V-based microarchitecture.

The rest of this paper is organized as follows. Section II discusses the related work. The preliminaries of Spectre attack are presented in Section III. In Section IV, the cache implementation details of BOOMv3 and Xuantie-910 along with their impact on security are introduced. The experimental procedure is reported in Section V. Finally, Section VI provides conclusions and suggestions for future work.

II. Related Work

In this section, we briefly introduce existing research on cache side-channel attacks.

A. Time-driven Cache Side-channel Attacks

Time-driven cache-based side-channel attacks exploit the inherent property of the difference in response time between cache hits and cache miss. Kesley [10] first proposed secret key analysis based on cache hit rates, revolutionizing traditional analysis methods. After then, a variety of effective cache-based side-channel attacks have been proposed. What is even more threatening is that these attacks can be implemented on real platforms and successfully obtain private data such as AES keys [11] and OpesnSSL random values [12]. The attack environment is gradually shifting from single-core to cross-core case and from microprocessor to cloud environment. This poses a serious threat to microarchitecture security. Research on cache-based side-channel attacks on x86, ADM and ARM architectures is relatively mature. However, there are still many challenges in reproducing cache-based side-channel attacks on RISC-V architectures, and there is also a lack of related research work.

B. Countermeasures for Cache-based Side-channel Attacks

Traditional caches use a fixed mapping policy that allows an attacker to easily locate the cache lines involved in a victim's program to run and perform operations such as probing or eviction. Several new caching architectures have been proposed to address this weakness. HybCache in [13], [14] used a random substitution strategy to prevent attackers from accurately evicting. F. Liu and R. B. Lee in [15] randomly loaded data as a way to avoid attackers launching reuse-based attacks against specific addresses. The sharing of hardware resources such as cache and branch predictors is the root cause of cache side channels. Isolation-based, encryption-based and other schemes have been proposed.

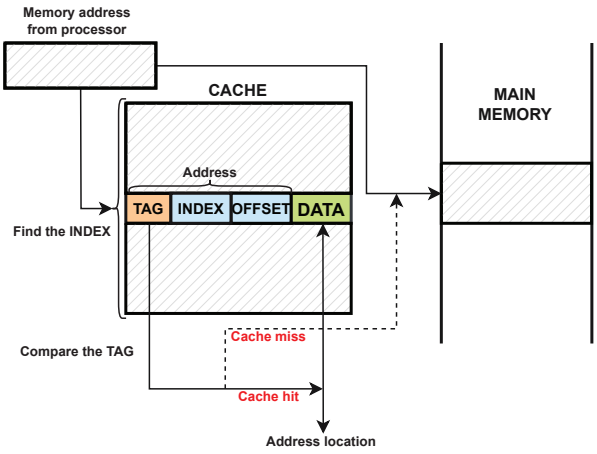

Fig. 1. Cache operation mechanism.

III. Background

In this section, we introduce RISC-V ISA, the structure and organizational form of cache and cache-based side-channel attacks including their principles and types. We also introduce Spectre attacks.

A. RISC-V ISA

RISC-V is a free and open-source ISA based on the principles of the Reduced Instruction Set Computing (RISC). Because the RISC-V instruction set is designed to be implemented considering both the high performance and low power consumption, it is not only concise but its different modules can be organized together in a modular way.

RISC-V consists of a required basic integer instruction set and various optional extensions. Extensions are denoted with a single letter, e.g. M (integer multiplication and division), A (atomic instructions), C (compressed instructions), etc. A comprehensive description of the RISC-V instruction set is available in [16].

B. Cache

The processing speed of CPU is too fast and the relatively slow access speed of the main memory forms a contradiction. Based on time and spatial correlation, the cache solution is brought up. The whole cache space is divided into cache lines. Each cache line is divided into Address and Data. Address is composed of Tag, Index and Offset. The Data part stores a piece of data with continuous addresses, while the Tag part stores the public addresses of these data [17]. Index is promoted to locate cache line. Offset is used to locate single byte in cache line. Those cache lines which share the same index are called set.

As shown in Fig. 1, when a cache client attempts to access data, it first checks cache. If the requested data is found there, it is called a cache hit. The percentage of attempts that result in a cache hit is called cache hit ratio. The requested data not found in cache (called a cache miss) is fetched from the

main memory and copied into the cache. How this is done and which data is popped from the cache to make room for new data depends on the caching algorithm, caching protocol, and system policy adopted.

C. Cache-based Side-channel Attacks

For the purpose of improving processor performance by predicting future program behavior, microarchitectural components maintain the state that depends on past program behavior and assume that future behavior is similar or related to past behavior. When multiple programs are executed on the same hardware, either concurrently or via time sharing, the changes of microarchitectural state caused by the behavior of one program may affect other programs. This, in turn, may result in unintended information leaks from one program to another [18]. Side-channel attacks using this principle were first proposed by Kocher et al. Over the years, side channels have been demonstrated over multiple microarchitectural components. However, in general, attackers are not able to observe each of the victim's cache accesses. Therefore, indirect observations are used. In some of these attacks, attackers measure the access time of each of their own individual memory accesses, after some interference with the victim. In other attacks, attackers observe the total execution time of the victim's security-critical operation, instead of the access time of each memory access [6]. The types of commonly-used cache-based side-channel attack include Evict-and-reload Attack[1] [8], Prime-and-probe Attack [19] and Flush-and-reload Attack [7].

D. Spectre Attacks

Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration through microarchitecture covert channels. Since security check takes a long time, processor will execute subsequent instructions in advance based on the prediction of the corresponding speculative mechanism until the security check result is available. This time period is the transient execution window. Transient instructions executed within this time window may calculate the unauthorized results on the prediction branch and then leak them through the microarchitecture convert channels. In such an attack, the attacker starts by locating or introducing a sequence of instructions within the process address space which, when executed, acts as a covert channel transmitter that leaks the victim's memory or register contents. The attacker then tricks CPU into speculatively and erroneously executing this instruction sequence, thereby leaking the victim's information over the covert channel. The above description of the Spectre attack is general and needs to be specifically instantiated with a method to induce erroneous speculative execution and a microarchitectural covert channel such as the cache-side channel described above.

[1]In the names of the listed representative attacks, the words before and after "-and-" represent that the attacker performs two actions, e.g., "evict" and "time", with the "-and-" indicating the attacker waits in-between these actions for the victim to perform the security-critical operation.

IV. Cache-related Security Analysis

A. Introduction of BOOMv3 and Xuantie-910

BOOMv3 is a synthesizable and parameterizable open-source RV64GC RISC-V core written in the Chisel hardware construction language. Xuantie-910 is a RISC-V compatible 64-bit high-performance processor developed by T-head Semiconductor Co., Ltd. It delivers industry-leading performance in control flow, computing and frequency through architecture and micro-architecture innovations.

B. Cache Implementation Details of BOOMv3

- **BOOMv3 data cache overview:** The size of L1 cache in BOOMv3 of a small configuration is 16K bytes, using a 4-way set-associative structure. The size of the cache line is 64 bytes, with a total of 64 sets. The L1 data cache uses virtual address index and physical address tag (VIPT). The cache line can be divided into four groups, each containing 128-bit data array. The data array adopts the structure of multiple banks, and there are a total of 8 banks. Each bank has a bit width of 32 bits. Each time up to four banks of data can be read, i.e., the maximum bit width of each read access is 128 bits.
- **Read strategy:** When cache performs a read operation, the cache arbiter will read all 4-way data in the set corresponding to the index part of the address, and compare the tag part of the address with the tag of each way in the set to determine whether there is a hit.
- **Write strategy:** When cache performs a write operation, the cache arbiter adopts a random replacement strategy to backfill the data to a certain way in the set corresponding to the index part of the address.

C. Cache Implementation Details of Xuantie-910

- **Xuantie-910 data cache overview:** The size of the L1 data cache of Xuantie-910 is 64K bytes, using a 2-way set-associative structure. The size of the cache line is 64 bytes, with a total of 512 sets. The L1 data cache uses physical address index and physical address tag (PIPT) with a first-in-first-out replacement strategy. The cache line can be divided into four groups, each containing 128-bit data. Addr[5:4] represents the fourth and fifth bits of the address to access the memory. Each array stores 128-bit data, so Addr[5:4] corresponds to the lower two bits of the index, and for a certain address, the location where the data mapped to the cache is stored can be determined.
- **Read strategy:** When cache performs a read operation, the index of accessing the low array is the access address itself, while the index of accessing the high array will invert the fourth bit of the access address, so as to read the data of the two ways, and then select the data of the channel according to the verification result of the tag.
- **Write strategy:** When cache performs a write operation, High Array and Low Array use the same index, and half of the data of the cache line can be written to the cache at a time. The lowest bit of the index is the way to be backfilled. If the way to be backfilled is way1, one needs

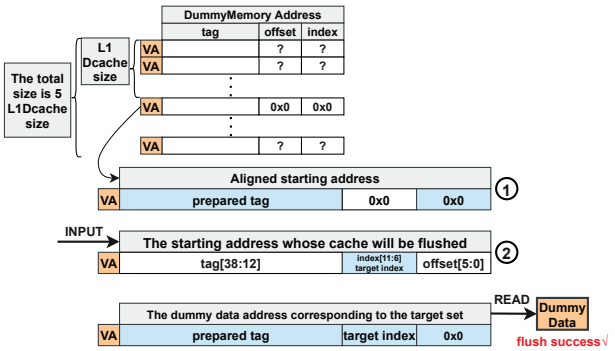

Fig. 2. Process of accurate flushing function.

to adjust the order of the data, and swap the end to end in 128-bit units. The penultimate bit of the index is cnt. Since a maximum of 256 bits can be written at a time, a cache line can be written only twice. The cnt bit is used to calculate whether to write the first half or the second half of the cache line.

D. The Impact of Cache Design Differences between BOOMv3 and Xuantie-910 on Security

- **Accurately flush method for BOOMv3 cache line:** BOOMv3's L1 Dcache has a total of 64 sets, and the cache line size is 64 bytes. It requires 6 bits in the address to match the set index of the cache arbiter, and the offset requires 6 bits. These 12 bits exactly correspond to virtual/physical address. Therefore, after the index part of the virtual address is converted by MMU, the index part of the exact same physical address is obtained. So if one wants to accurately flush all cache lines in a set, it only needs to obtain the index of the set, and then access the garbage with the same index multiple times. The address where the data is located is sufficient. The multiple accesses here are required because the dcache of BOOMv3 adopts a random replacement strategy. The way to be written each time is to fill in one of the four ways at random. Repeated flushing can increase the eviction rate. For a certain location in continuous memory the one-time eviction process is shown in Fig. 2.
- **The same flush method on Xuantie-910:** Since the L1 Dcache of Xuantie-910 has a total of 512 sets, the cache line size is 64 bytes, the number of bits in the address used to match the set index of the cache arbiter requires 9 bits, the offset requires 6 bits, and a total of 15 bits are required. However, according to the RISC-V standard only the first 12 bits of the lower virtual address can be used as the offset of the page table where the virtual address is translated into the physical address, which corresponds to 4,096 bytes of the page table page. The physical memory address is 56 bits, of which 44 bits are the physical page number. The remaining 12 bits are completely inherited from the virtual address, so the [14:12] bits of the virtual address are different from the [14:12] bits of the physical address. When we flush the data cache mapped to the memory where the specified data is located, we cannot locate the corresponding set index is retained and the tag is modified to flush the same cache line.

```
if (idx < array1_sz) {
        dummy = array2[array1[idx] * L1_BLOCK_SZ_BYTES];
}
```

Fig. 3. Spectre V1 victim function.

V. Experimental Procedure

A. Cache-based Side-channel Attack Flow

Flush-and-reload is adopted in the current attack method to construct cache based side channel of the leaked secret byte. Let C denote the content stored in a target memory address A, and C has been loaded into cache. The basic principle of the flush-and-reload technique can be explained by the following 3 phases:

- **Phase 1:** The attacker flushes C from cache.
- **Phase 2:** The attacker waits for the victim's code to access A. This access will load C back into cache.
- **Phase 3:** The attacker re-accesses A and measures the time of this access. If the access time is within a certain threshold, it means that the victim has accessed A, which can lead to the leak of the secret byte.

In the above flush-and-reload operation steps, it is relatively simple to load memory data into cache, while difficult to flush cache, because the current RISC-V ISA does not support the direct flush operation of cache line, in contrast to X86-based ISA which has *clflush* interface to achieve flush operation [20]. The cache related operations in RISC-V have been defined in the newly proposed Cache Management Operation (CMO) extension, which, however, may take a long time to improve and implement. Therefore, we cannot directly flush the specified cache address at present, but it is quite necessary to flush cache accurately to realize various attacks such as Spectre using the cache based side channel.

Taking the representative Spectre V1 that exploits the cache-side channel to leak secret data as an example, the victim function code of the attack is shown in Fig. 3. To implement the Spectre V1 attack, the attacker needs to create three conditions.

- Firstly, the value of the *idx* shown in Fig. 3 is maliciously chosen (out-of-bounds), so that *array1*[*idx*] resolves to a secret byte *k* somewhere in the victim's memory, i.e., *idx = (address of a secret byte to read) − (base address of array1)*.
- Secondly, *array1_sz* and *array2* are uncached, but *k* is cached, thus making the branch condition wait for uncached parameters. It may take more time to determine the branch result. When the branch prediction is true, the processor will go to access *array2*[$k \times 4096$], and at this

time, *k* is in the in the cache so it can return immediately, which provides sufficient transient execution window for the processor to execute the instructions in the conditional branch statement to leak the secret byte *k* into the cache.
- Finally, the *idx* received by the previously executed conditional branch statements were valid, leading the branch predictor to assume the "if" statement will likely be true.

In order to achieve the above attack conditions and build the side channel, the flush function that could accurately flushes the cache, i.e., flushes the specified cache address, is to be implemented. We need a workaround to achieve the same effect as the *clflush* operation. For example, we can overwrite the original content that we want to flush by loading new dummy content to the specified location in the cache, essentially a tag changes in the cache, thus we disguise the flush equivalent operation to the specified address in the cache. As for an extreme example, if the entire cache has 64K bytes, and when 64K bytes of random memory data (dummy data) is loaded into an array, this data will be loaded into the data cache at the same time. Then all the original data in cache will be flushed away. The approach we take is a similar one, but we only flush out the cache units we care about, not the entire cache.

B. Simulation Environment Setup

In the experiment against a BOOMv3 side-channel attack, we used the Chipyard [21] working environment developed by UC Berkeley to complete the Spectre V1 attack simulation experiment of BOOMv3 and successfully reproduced it in FPGA. Next, we built a QEMU-based Xuantie-910 verification environment to verify whether the precision flush function is available on Xuantie-910.

C. Cache-based Side-channel Attack on BOOMv3

The implementation idea of the flush function in our spectre attack code is as follows:

We first pass the starting address of the data in main memory that needs to be flushed in its corresponding cache and the data length *sz* (in bytes) to be flushed as entry parameters into the function. Then we use the virtual address to calculate the number of cache sets that need to be flushed to flush *sz*-sized data, and the specified memory address of each set that needs to be flushed corresponds to the index part of the cache arbiter. Finally, according to the calculation result, the cache set at the specified location is filled with dummy data to flush out the content. It should be noted that because the BOOMv3 cache adopts a random replacement strategy by default, it requires more flushes. The specific idea is the same as Fig. 2 in Section IV.

In order to prove that our flushing function is effective, we measured the cycles of loading data from cache and RAM (Random Access Memory) before using the flushing function, and the cycles of loading data from cache and RAM after using the flushing function in the experimental platform to see if they are what we expect.

```
iteration=1, read from ram, cycles = 72
iteration=1, read from data cache, cycles = 39
iteration=1, flush and then read from ram, cycles = 62
iteration=1, read from data cache after flush, cycles = 40

iteration=2, read from ram, cycles = 70
iteration=2, read from data cache, cycles = 39
iteration=2, flush and then read from ram, cycles = 62
iteration=2, read from data cache after flush, cycles = 40

iteration=3, read from ram, cycles = 72
iteration=3, read from data cache, cycles = 39
iteration=3, flush and then read from ram, cycles = 62
iteration=3, read from data cache after flush, cycles = 40

iteration=4, read from ram, cycles = 70
iteration=4, read from data cache, cycles = 39
iteration=4, flush and then read from ram, cycles = 62
iteration=4, read from data cache after flush, cycles = 40
```

Fig. 4. The result of accurate cache flushing function on BOOMv3.

```
================ This is a POC of spectre_v1 (Branch Condition Check Bypass)
the secret key is:!"#ThisIsTheBabyBoomerTest
m[0x0x80002708] = want(!) =?= guess(hits,dec,char) 1.(9, 33, !) 2.(2, 94, ^)
m[0x0x80002709] = want(") =?= guess(hits,dec,char) 1.(10, 34, ") 2.(5, 255, )
m[0x0x8000270a] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(4, 255, )
m[0x0x8000270b] = want(T) =?= guess(hits,dec,char) 1.(9, 84, T) 2.(5, 255, )
m[0x0x8000270c] = want(h) =?= guess(hits,dec,char) 1.(8, 104, h) 2.(5, 255, )
m[0x0x8000270d] = want(i) =?= guess(hits,dec,char) 1.(7, 105, i) 2.(3, 255, )
m[0x0x8000270e] = want(s) =?= guess(hits,dec,char) 1.(7, 115, s) 2.(3, 255, )
m[0x0x8000270f] = want(I) =?= guess(hits,dec,char) 1.(7, 73, I) 2.(5, 255, )
m[0x0x80002710] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(6, 255, )
m[0x0x80002711] = want(T) =?= guess(hits,dec,char) 1.(7, 255, ) 2.(6, 84, T)
m[0x0x80002712] = want(h) =?= guess(hits,dec,char) 1.(7, 104, h) 2.(3, 255, )
m[0x0x80002713] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(3, 255, )
m[0x0x80002714] = want(B) =?= guess(hits,dec,char) 1.(5, 66, B) 2.(5, 255, )
m[0x0x80002715] = want(a) =?= guess(hits,dec,char) 1.(8, 97, a) 2.(5, 255, )
m[0x0x80002716] = want(b) =?= guess(hits,dec,char) 1.(7, 98, b) 2.(4, 255, )
m[0x0x80002717] = want(y) =?= guess(hits,dec,char) 1.(8, 121, y) 2.(2, 255, )
m[0x0x80002718] = want(B) =?= guess(hits,dec,char) 1.(8, 66, B) 2.(5, 255, )
m[0x0x80002719] = want(o) =?= guess(hits,dec,char) 1.(7, 111, o) 2.(3, 255, )
m[0x0x8000271a] = want(o) =?= guess(hits,dec,char) 1.(6, 111, o) 2.(3, 255, )
m[0x0x8000271b] = want(m) =?= guess(hits,dec,char) 1.(7, 255, ) 2.(5, 109, m)
m[0x0x8000271c] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(2, 153, )
m[0x0x8000271d] = want(r) =?= guess(hits,dec,char) 1.(9, 114, r) 2.(2, 25, )
m[0x0x8000271e] = want(T) =?= guess(hits,dec,char) 1.(8, 84, T) 2.(7, 255, )
m[0x0x8000271f] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(6, 255, )
m[0x0x80002720] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(4, 255, )
m[0x0x80002721] = want(t) =?= guess(hits,dec,char) 1.(6, 116, t) 2.(6, 255, )
```

Fig. 5. The result of accurate cache flushing function on BOOMv3.

It can be seen from Fig. 4 that after the data is loaded into the cache and then flushed, the number of cycles of loading the data is the same as the cycles of the initial loading from the RAM, which proves that the flushing function is effective. After that, we apply this function to build a cache side channel in the Spectre V1, V2, V4, and V5 attack codes, and successfully read the secret data on BOOMv3. The experimental results are shown in Fig. 5, which further prove the effectiveness of the flushing function.

D. Cache-based Side-channel Attack on Xuantie-910

In order to prove the impact of the different designs of Xuantie-910 cache and BOOMv3 cache on side-channel attacks, we implement the cache flushing function in Xuantie-910 in the same way as in BOOMv3. We conduct experiments in the Xuantie-910 simulation platform, and measure the number of cycles of loading data from cache and RAM before and after flushing using this function to see if it meets expectations. As a comparison, we have written a function to flush the entire cache of Xuantie, that is, when loading a random memory data (dummy data) of size 64K bytes into an array, these data will be loaded into Xuantie-910 data cache at the same time, and thus all the original data in the cache will be flushed. The results of the two cache flushing functions are shown in Fig. 6 and Fig. 7, respectively.

```
iteration=5, read from ram, cycles = 322
iteration=5, read from data cache, cycles = 162
iteration=5, flush and then read from ram, cycles = 186
iteration=5, read from data cache after flush, cycles = 256

iteration=6, read from ram, cycles = 326
iteration=6, read from data cache, cycles = 162
iteration=6, flush and then read from ram, cycles = 188
iteration=6, read from data cache after flush, cycles = 258

iteration=7, read from ram, cycles = 328
iteration=7, read from data cache, cycles = 174
iteration=7, flush and then read from ram, cycles = 180
iteration=7, read from data cache after flush, cycles = 256

iteration=8, read from ram, cycles = 328
iteration=8, read from data cache, cycles = 168
iteration=8, flush and then read from ram, cycles = 174
iteration=8, read from data cache after flush, cycles = 254
```

Fig. 6. The result of accurate cache flushing function on Xuantie-910.

```
iteration=5, read from ram, cycles = 244
iteration=5, read from data cache, cycles = 118
iteration=5, flush and then read from ram, cycles = 240
iteration=5, read from data cache after flush, cycles = 116

iteration=6, read from ram, cycles = 266
iteration=6, read from data cache, cycles = 120
iteration=6, flush and then read from ram, cycles = 270
iteration=6, read from data cache after flush, cycles = 118

iteration=7, read from ram, cycles = 282
iteration=7, read from data cache, cycles = 96
iteration=7, flush and then read from ram, cycles = 196
iteration=7, read from data cache after flush, cycles = 90

iteration=8, read from ram, cycles = 198
iteration=8, read from data cache, cycles = 92
iteration=8, flush and then read from ram, cycles = 310
iteration=8, read from data cache after flush, cycles = 92
```

Fig. 7. The result of entire cache flushing function on Xuantie-910.

Comparing the above results and the experimental results of BOOMv3, it can be seen that the flush function that accurately flushes the cache does not successfully flush the data on the Xuantie-910, which further verifies our conjecture about the impact of the design of the Xuantie-910 cache on the security of the micro-architecture.

VI. Conclusion and Future Work

This paper explored the impact of cache design on microarchitectural security, using two representative RISC-V-based high-performance processor cores, i.e., BOOMv3 and Xuantie-910, as representatives. Given the lack of cache flushing instructions in the RISC-V-based architecture, we created attack conditions and constructed side channels by loading extraneous data into specific cache lines. Then we compared the cache flushing experiments over Xuantie-910 and BOOMv3. The results show that the accurate flushing of specified cache, which can be achieved on BOOMv3 with VIPT memory-mapped cache mechanism, cannot be accomplished on Xuantie-910 with PIPT memory-mapped cache mechanism, which efficiently decreases the possibility of flush-and-reload attack on Xuantie-910. Although the original intention of Xuantie-910 adopting PIPT is not for avoiding cache-based side-channel attack, such a design objectively prevents the precise addressing of specified cache lines. On the other hand, it is well-known that the addressing efficiency of PIPT is lower that that of VIPT. This provides us with insight that we can adopt a VIPT-based cache with a mechanism designed from hardware level for preventing the accurate access of cache line index, which can not only keep the high-performance addressing of VIPT but also improve chip security.

Acknowledgment

The authors would like to thank Dr. Hongyu Wang for his constructive suggestions, and great help and support.

References

[1] T. Lu, "A survey on risc-v security: Hardware and architecture," *arXiv preprint arXiv:2107.04175*, 2021.
[2] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen *et al.*, "Xuantie-910: Innovating cloud and edge computing by risc-v," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–19.
[3] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020.
[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.
[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.
[6] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.
[7] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," 2014, pp. 719–732.
[8] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.
[9] A.-T. Le, B.-A. Dao, K. Suzaki, and C.-K. Pham, "Experiment on replication of side channel attack via cache of risc-v berkeley out-of-order machine (boom) implemented on fpga," in *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, 2020.
[10] O. Acıicmez, W. Schindler, and C. K. Koc, "Cache based remote timing attack on the aes," in *Topics in Cryptology – CT-RSA*. Springer, Berlin, Heidelberg, 2006, pp. 271–286.
[11] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," vol. 23, pp. 37–71, 2010.
[12] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack," 2014.
[13] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments." USENIX Association, 2020, pp. 451–468.
[14] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu," in *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.
[15] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.
[16] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.
[17] A. Rousskov and D. Wessels, "Cache digests," *Computer Networks and ISDN Systems*, vol. 30, no. 22-23, pp. 2155–2168, 1998.
[18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.
[19] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.
[20] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, 2011.
[21] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.