Replicating Side Channel Attacks on RISC-V

Ben Chen
Southern University of Science and Technology
Shenzhen, Guangdong, China
chenb2022@mail.sustech.edu.cn

Jikun Liao
Southern University of Science and Technology
Shenzhen, Guangdong, China
12212224@mail.sustech.edu.cn

Shuwei Zhang
Southern University of Science and Technology
Shenzhen, Guangdong, China
12212912@mail.sustech.edu.cn

ABSTRACT

Microarchitecture vulnerability is unclear on open-source RISC-V processors due to the agile development. However, the open-sourceness of these IPs facilitates the analysis of the side-channel vulnerabilities like Spectre and Meltdown. It is crucial to identify these flaws in the pre-silicon stage [4]. Additionally, we have found some recent state-of-the-art attacks that have not been validated on RISC-V processors.

With this initiative, we conducted a comprehensive research on side-channel attacks, and replicated them on X86-64 chips. Next, we analyzed the public documents and source code to address the vulnerable components. Using an open-source RISC-V processor OpenXiangShan by simulation and FPGA, we migrated and re-implemented three attack primitives in our study, Spectre[8], Phantom[12] and Downfall[9].

KEYWORDS

Side-channel attack, Cache-based side channel, Microarchitecture vulnerability

1 INTRODUCTION

Side-channel attacks are a type of security exploit that leverage the physical characteristics of a system to extract sensitive information. Microarchitecture vulnerabilities include cache-based side channel, interrupt-based side channel, microarchitectural data sampling, etc. Recent studies have demonstrated the evolving complexity and efficacy of side-channel attacks. However, these findings were performed on X86-64 or ARM processors. However, the OpenXiangShan RISC-V processor has yet to be exposed to valid exploitation. In our survey, the mitigation of side-channel implemented on RISC-V claimed that their defense was effective, but did not provide a valid proof-of-concept attack for their experiment. Therefore, before finding a defensive methodology, our first objective is to find a valid attack surface.

In this project, we investigated the world of various side-channels with the attack surfaces being the classical Specter, and brand-new Phantom and Downfall attack. Meanwhile, we also surveyed several related attack surfaces exploiting the architectural component like TLB, interrupt, and buses. However, due to the limitation of time, we are only able to leave them to the future works.

Spectre attack exploits the speculative execution of modern processor to leak data by misleading the branch predictor and triggering the misprediction to accidentally bring sensitive information to cache. Phantom attack is able to observe the performance counter to induce the instruction decoded at the frontend of processor, which causes a information leakage. The Downfall attack takes advantage of the vector instruction to rapidly fill out the internal buffer (also serves like a cache) and sequently conducts a microarchitectural data sampling attack.

The main challenges we encountered are that, firstly we are new to microarchitectural vulneraibility, which took us weeks to getting started. Secondly, the ecosystem of open-source processors remains incomplete and most of them are not currently commercially-off-the-shelf. So setting up the experiment workflow is not so easy as that in x86 or ARM processors, because it often requires building the simulator and preparing the custom workloads with a complicated procedure. For now, we've not yet overcome them and are still validating our attack experiment.

2 BACKGROUND

2.1 Side Channel Attacks

In contrast to the mathematical models used in cryptanalysis, side-channel attacks target at the actual hardware systems implementing the cryptographic algorithms. These attacks exploit certain characteristics of specific hardware systems and carry out physical attacks on devices to steal secret from them. Compared with direct attack, SCAs are more powerful but less accurate.

2.2 Cache

Cache is a smaller, faster memory component located closer to the core, designed to store copies of frequently accessed data from the main memory. The principle of cache is temporal and spatial locality of data. This means that the data accessed will probably be accessed again and the neighbour data will probably be accessed in the future.

Typically, a modern CPU has one isolated L1 cache for each core and shared L2 cache within a bundle of core and a global L3 cache. Most of the cache hierarchy design is inclusive, ensuring that data in lower cache is in higher cache as well.

2.3 Speculative Execution

Modern processors are equipped with pipeline and superscalar technology. However, when dealing with a branch instruction, processors know the result cycles after and within this window, it has to stall. To ensure both correctness and efficiency, branch prediction and speculative execution engine must also be accompanied. By this technique, processor predicts the next branch and executes parts of the branch before obtaining the true branch result. If the prediction fails, it will roll back and discard the speculative result.

2.4 Cache-based Side Channel Attacks

Cache side-channel attacks leverage the differences in access times between cache hits (when data is present from the cache) and cache

misses (when data is retrieved from the main memory). By measuring these access times, attackers can infer sensitive information about the data being processed. This flaw are often exploited with other flaws such as prefetcher and speculative execution.

Cache attacks are generally categorized into four kinds, which are flush+reload, flush+flush, prime+probe and evict+time. These are often require a high-resolution timer, while several state-of-the-art attacks leverage the new x86 instructions without a timer such as MWAIT[14].

2.5 Microarchitecutral Data Sampling

As further optimization, modern processors leverage small-sized internal cache inside microarchitectural components. For example, in the Load-Store Unit (LSU), it will try to fully occupy the bandwidth of memory transmission by adding a committed-store buffer. The store instruction is committed in batch to the memory when the buffer reaches the batch size. Similar to cache, when instruction accesses data inside the buffer, it can fetch data directly from the buffer, which also causes a timing difference. As a result, an attacker at the same CPU core could steal data from these shared buffers.

2.6 OpenXiangShan

In 2019, the Institute of Computing Technology of the Chinese Academy of Sciences initiated the XiangShan high-performance open-source RISC-V processor project. It provides one of the world-wide highest-performing open-source RISC-V processor cores IP with a agile development workflow, similar to Berkeley-Out-of-Order-Machine.

Ghaniyoun et al.[4] presented their automatic framework in finding microarchitectural flaws in XiangShan, but the primitive is not well-developed into a valid attack.

3 RELATED WORK

Gerlach et al.[3] performed a systematic analysis of microarchitectural components in two commercially-off-the-shelf RISC-V processors T-Head C906 and SiFive U74. They discovered the available high-resolution timer, cache capacity and replacement policy and branch predictor specification in them. With this in hand, the attacker can perform an attack on the Instruction Cache (ICache). If a victim has a secret-dependent branch, the attacker can jump directly to the two snippets of the if-else and measure the clocks difference in executing them. Since the executed instruction is present in ICache, it will be executed faster by the attacker. Based on the observation, the attacker is able to infer the secret bit. Similarly, it can be done without executing the victim's code. The attacker registers a fault handler with recording the end time, and maliciously sets the register bank that will trigger faults in victim's code, and finally jump to the code to immediately trigger the fault.

4 SPECTRE

In 2018, Gruss et al.[8] proposed the new Spectre attack on Intel/AMD processors. The root cause of Spectre and its variants lies on the implicit state transformation in cache. When an instruction is executed speculatively due to the branch prediction, it's expected that all states before the speculation will be resumed. However, the discard of store-load instruction will not roll back the states in cache, indicating that the data speculatively brought to cache will stay in cache. This causes information leakage if the data is sensitive.

The attack is verified on x86 and ARM machines. However, the attack depends on several primitive components in architecture such as the timer, cache and branch predictor. And yet there's no valid proof-of-concept attack on XiangShan. Our goal is to develop the Spectre attack primitive on XiangShan core.

4.1 Spectre on x86 and ARM

The Spectre attack assumes that the attacker and victim shares the same cache but in different core. The victim has a similar code snippet like

```
uint8_t array[160] = {1, 2, ..., 16};
char* secret = "Secret goes here!";

void victim_function(size_t x) {
   if (x < array1_size) { // array1_size = 16
       temp &= array2[array1[x] * CACHELINE_SIZE];
   }
}
```

Listing 1: Vulnerable code in victim

The attacker has access to `array2` and then train the branch predictor with legal input `x`. After sufficient training, the branch predictor will always predict the branch in victim function to be taken. So next time a caller invokes the victim function, the speculative engine will try to execute instructions inside if statement, even if it's out of bound.

```
for (int x = 0; x < ENTRY_SIZE; ++x) {
    victim_function(x);
}
flush_cache(array2);
victim_function(secret[i++]);

for (int i = 0; i < 256; i++) {
    addr = &array2[i * CACHELINE_SIZE];
    time1 = __rdtscp(&junk); /* READ TIMER */
    junk = * addr; /* MEMORY ACCESS TO TIME */
    time2 = __rdtscp(&junk) - time1;
    /* READ TIMER & COMPUTE ELAPSED TIME */
    if (time2 <= CACHE_HIT_THRESHOLD)
        results[i]++;
}
```

Listing 2: Training the branch predictor

To carry out the leak information, an attacker often adapts a probe array, i.e., `array2` in this example. The array is used to detect which element is loaded into cache, because the secret information is encoded to be the offset. A detail goes here: the offset accessed in `array2` must be a multiple of `CACHELINE_SIZE` because if the

offset is less, we are blinded to it. Adjacent data might be mapped to the same cache line and therefore, we are not able to tell that difference.

In real-world scenario, address of secrets can be in the kernel and attacker has to look for vulnerable snippets shown in Listing 1.

4.2 Spectre on XiangShan

The Spectre seems to work on XiangShan, and only a few tweaks need to be made to migrate. However, the branch predictor technology has advanced for a great leap in these years.

4.2.1 Branch Predictor. Nanhu, the 2[th] architecture of XiangShan, has μBTB, BTB, TAGE, RAS and Loop Predictor integrated in its branch predictor.

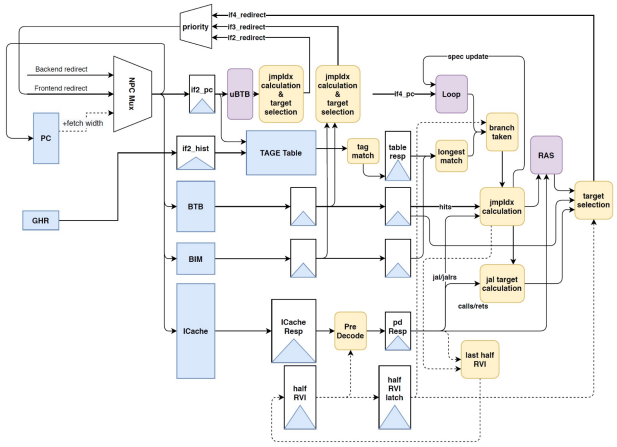

Figure 1: Branch predictor design of Nanhu

Among the components, TAGE with Loop Predictor is the top concerned for disabling the traditional Spectre attacks. TAGE is a global predictor with context information hashed into a tag. Most time it cooperates with loop predictor to enhance the accuracy, since most of the branches are actually the exit conditon of a loop. If the predictor can recognize that loop, it can skip the meaningless prediction.

However, this feature mitigates Spectre in Listing 2, by recognizing the loop and calling context in the training procedure and predicts that, in the next line when attack accesses the out-of-bound, the branch result is false.

4.2.2 Cache. Table 1 shows the default parameters of Nanhu architecture. Obviously, we need to tune the parameter in the above listing to the numbers shown below.

In cache eviction, we needs to create 8 trash variable with the same index (the bits from 7th to 14th), and the `CACHELINE_SIZE` is set to be 64. Meanwhile, the L1 sets number ensures that all ASCII character can be encoded and VIPT (Virtual Index Physical Tag) cache alias indicates that virtual address is enough for our exploitation.

4.2.3 Attack on Nanhu. Based on the discovery, we tweaks our exploitation to the listing shown below

Parameter	Value
L1 Sets	256
L1 Ways	8
L1 Linesize	64 bytes
Cache alias	Virtual Index Physical Tag

Table 1: L1 Cache parameters

```
for (int i = 0; i <= ENTRY_SIZE; ++i) {
    int x = i < ENTRY_SIZE ? i : secret[i++]
    flush_cache(array2);
    victim_function(x);
}

for (int i = 0; i < 256; i++) {
    addr = &array2[i * CACHELINE_SIZE];
    time1 = rdcycle(); /* READ TIMER */
    junk = * addr; /* MEMORY ACCESS TO TIME */
    time2 = rdcycle() - time1;
    /* READ TIMER & COMPUTE ELAPSED TIME */
    if (time2 <= CACHE_HIT_THRESHOLD)
        results[i]++;
}
```

Listing 3: Spectre on Nanhu

The major change is that we combine the final access to secret to the training. Actually, it is not the real implementation, since the second line involves a branch, but it could be optimized into a table array, which is trivial. Another change is the access to timer, in RISC-V ISA, the instruction to read a timer is `rdcycle`.

4.3 Mitigation

So far, most of the mitigation techniques are validated on RISC-V. Khasawneh et al.[7] proposed a temporary buffer called SafeSpec, to store the data before the instruction commits. To avoid another timing difference in that buffer, SafeSpec won't forward its internal data, so as to disable cache-based side channel attacks. However, this introduces 5 percent performance overhead and more circuits in chips.

An improvement was present by SpectreGuard [2] and SpecTerminator [6]. These works proposed techniques to detect possible vulnerable code snippet and transient window, and then mark with tags to prevent them from being loaded into cache before committing instruction.

5 PHANTOM

The latest research [12][15] shows that modern x86 processors make predictions very early in the pipeline, even before the current instruction is decoded. Although this aggressive mechanism could improve CPU performance, it leads to new risks that almost all types of instruction could be trained to trigger prefetch [15] and even speculative execution[12] within the transient window between

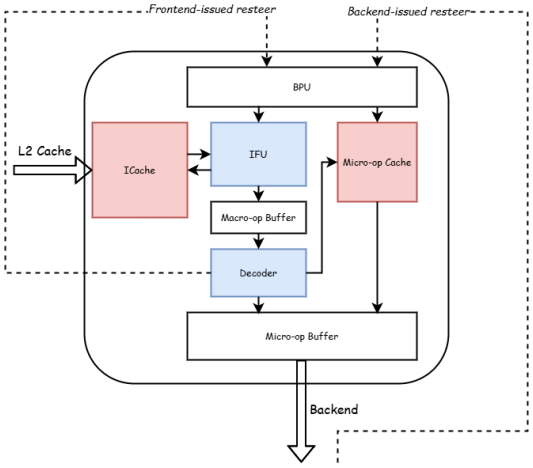

Figure 2: x86 Decode pipeline

prefetch and frontend resteer. This new source of speculation results in a broader attack surface and thus has significant research value.

Our goal is to investigate whether this issue also exists on RISC-V processors, e.g. XiangShan, as well as the potential attack methods based on this issue.

5.1 Phantom on x86

5.1.1 Instruction decode pipeline. Figure 2 shows a common frontend design in x86 processor. Firstly, Branch Prediction Unit(BPU) predicts next instruction's PC and issue a instruction fetch request to Instruction Fetch Unit(IFU). IFU then fetches corresponding instructions from ICache and dispatchs them to Decoder for uop translation. Finally, these uops are sent to backend to be executed.

In the entire frontend pipeline, BPU plays a crucial role. For branch instructions that depend on the execution results from the backend to determine the jump address, the branch predictor predicts the jump address and sends a request to the IFU, prefetching the predicted instruction into the pipeline. Furthermore, to reduce the delay between instruction fetch and instruction decode, many branch predictors predict whether the current instruction is a branch instruction and also predict the address of the next instruction even before the current instruction is decoded.

In this design, there are two main sources of mispredictions: the mismatch of predicted instruction types, such as predicting a non-branch instruction as a branch instruction, or predicting a direct jump instruction as a branch/indirect jump instruction; and incorrect prediction of the branch instruction's behavior, such as whether a branch is taken or not, or the address of an indirect jump. The former can be detected during frontend instruction decoding, while the latter depends on the execution results of the relevant instructions in the backend.

When a misprediction is detected, a resteer signal is sent to the BPU, and the instruction pipeline is flushed (depending on the specific architecture design). The BPU then returns to the point where the misprediction occurred, corrects the mistake, and resumes pipeline.

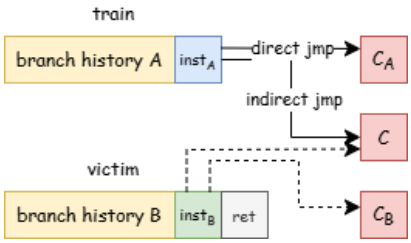

Figure 3: Phantom Workflow

While the execution-stage detectable misprediction often leaves a long enough transient execution window for subsequent instructions, which is exploited by the famous Spectre attack[8], recent research[12] shows that the relatively short transient window caused by frontend misprediction could also make subsequent instructions enter the IF, ID, or even instruction execution.

5.1.2 Observation Channels. In order to better understand the frontend pipeline behavior of the instruction prefetch-decode stage and lay the ground for future work on RISC-V processors, we reproduced the experiments from the paper[12] on AMD Zen2 and Zen3 processors. Due to the lack of open-source code from the original paper, we referred to several related works and leveraged our own understanding to develop the code. Additionally, there are some differences in the implementation design compared to the original paper.

The core of the experiment is to test the transient window length caused by mispredictions triggered by different instruction combinations set up for training/victim execution. This requires building up proper observation channels to track the advance of subsequent instructions through the pipeline. More specifically, we observe the progression of instructions through three stages in the pipeline: IF, ID, and EX.

The generic procedure is as follows. We have two snippets of code A and B, where A ends with a training branch to C. Branch A is either a direct jump/conditional jump instruction, where the target address is determined by a relative offset, or indirect jump instruction or return instruction, where the target address is specified by an absolute address in a register. For the former, we place code segment C_B at the same offset as victim instruction B. We want to observe that running B after multiple executions of A causes a misprediction to C_B / C that leads to detectable signals in IF, ID and EX observation channels.

IF Channel

We use Flush+Reload method [13] to observe whether a instruction is brought into ICache.

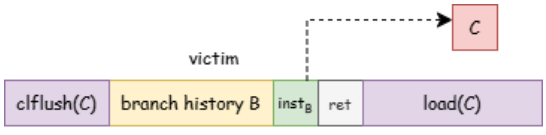

Figure 4: IF Channel

Before testing B, we use the `clflush` instruction to flush the content at the misprediction address C from both the L1 and L2 caches. Next, we execute code snippet B to trigger the mis-branch to C. We then measure the time it takes to load C. A relatively short access time for C indicates that C's instruction has been prefetched into the L1-ICache, which serves as the IF channel.

ID Channel

Building ID Channel is a bit of tricky, since we cannot directly manipulate decoder and μop cache, since micro-op cache is independent of the memory subsystem. Alternatively, we use hardware performance counters to monitor instructions dispatched into μop cache.

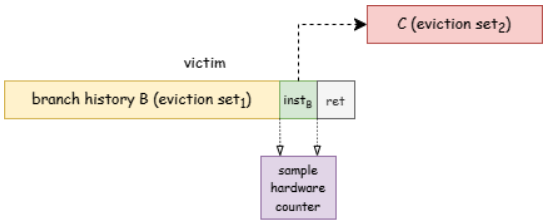

Figure 5: ID Channel

As method provided in [12], we choose *de_dis_uops_from_decoder.de_dis_uops_from_opcache* on Zen2 and *op_cache_hit_miss.op_cache_miss* on Zen3. Follow the procedure illustrated in figure 5, we first train A to jump to C which contains a series of direct jump instructions that will fill a certain μop cache set using method in [10]. Then we execute B whose branch history also includes a direct jump series that maps to the same μop cache set as C maps to, which will replace the μop cache occupied by C. Finally, we sample the performance counter before and after B's execution. If a misprediction to C enter ID stage, it will result in eviction of one or more ways of the set, which is detectable using performance counters. By comparing the results with and without training, we can decide whether a misprediction occurred or not. In addition, on Zen2 we can directly use *de_dis_uops_from_decoder.de_dis_uops_from_both* to monitor the differences in the number of uop instructions issued to backend to determine whether C's instructions are decoded.

In practice, the eviction jump series consists of 3 15 direct jumps separated by 2048 4096 bytes depending on the μop cache architecture.

EX Channel

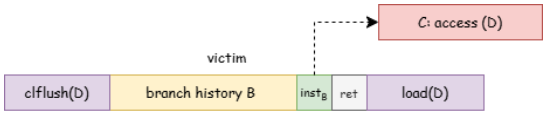

Figure 6: EX Channel

To observe whether the transient window is long enough for instruction execution, we use a similar method in IF but by monitoring a memory access operation instead. That is, flush a cache line mapped to address D, execute B which might trigger a misprediction loading D from memory, and test the time to access D to determine whether a transient load occurred.

5.1.3 Triggering Misprediction. To trigger a misprediction on B by training A, we must set them in the same branch history context, thus resulting in BTB (Branch Target Buffer) index aliasing. We follow the reverse engineering in previous work [11], generating training address and victim address in pairs by flipping certain bits to create BTB collision. For AMD Zen2, we choose to flip the 19th bit and 31st bit, while for Zen3 21st bit and 33rd bit. Since AMD Zen2 and Zen3 use a relatively short branch history compared to Intel, 4 8 direct jumps are enough to craft a valid history.

5.2 Phantom on RISC-V

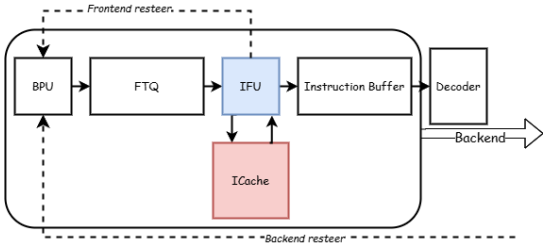

Figure 7: Nanhu Frontend

5.2.1 Nanhu Frontend. As a RISC-V processor, the frontend of Xiangshan has two major differences compared to the typical x86 frontend:

- Due to the fixed instruction length (4 bytes or 2 bytes for compressed instructions), the instruction decoding speed in RISC-V is relatively fast, so there is no need for a μop cache to accelerate decoding.
- Information such as whether the instruction is a branch instruction and the jump address can be obtained in the immediately after fetching the instruction through pre-decoding. Branch prediction checks are directly integrated into the IFU module.

As a result, once an instruction successfully enters the instruction buffer, it can be considered as being issued to the backend for execution.

Besdides, the open-source documentation of Xiangshan mentions that when a branch prediction error is detected, in addition to sending a redirection signal to the BPU, the entire IFU pipeline is flushed. This may result in a smaller transient window.

Therefore, our goal is to explore how this architectural change affects the transient window after a misprediction, as well as the instruction combinations that lead to mispredictions. Additionally, we aim to investigate whether this new architecture presents any potential issues.

5.2.2 Observation Channels. To determine how far an instruction advance in the pipeline, we select the IFU and Decoder as observation channels.

Although it is difficult to directly observe through methods like manipulating the cache or intervening the instruction pipeline, Xiangshan's rich set of frontend performance counters allows us to indirectly obtain a amount of information on instructions' status.

Furthermore, since Xiangshan's code is fully open-source, we can directly modify the code to output the events we want to observe.

The training and testing process is similar to that of x86. Here, we will only introduce the observation metrics used.

IFU

For IFU, we mainly observe two events. Whether a instruction is brought into ICache and whether the IFU flush signal is triggered. The latter indicates that the misprediction is detected during the pre-decoding stage.

The corresponding registers are *frontend_icache_miss_cnt* and *frontend_flush*.

Decoder

We determine whether an instruction will be issued to the back-end by monitoring the state of the decoder using *ctrlblock_decoder_util--ization*, which records the total number of instructions decoded over a period of time.

The purpose of setting up this channel is to determine whether any instructions continue to be sent to the decoder during the period between the detection of a misprediction and the complete flush of the pipeline.

EX

We use same method to monitor instruction execution as mentioned in x86.

However, due to the significant noise in the hardware counters, simply using polling to sample may lead to discrepancies between the observed results and the actual situation[1]. A more comprehensive test design is needed to ensure the reliability of the results.

5.2.3 Triggering Misprediction. As shown in figure 1, Xiangshan adopts a multi-level branch prediction design. Hence, to trigger a misprediction, we may need to mislead all levels of the predictor simultaneously.

Fortunately, perhaps to simplify the design, Xiangshan does not perform XOR hashing on the instruction's PC like Intel and AMD. Instead, it simply uses the lower bits of the PC XORed with the folded global history as the tag for the prediction table. To be more specific, μFTB uses PC's lower 16 bits as tag; TAGE-SC uses at most PC's lower 11 bits; FTB uses PC's lower 29 bits(9 for index and 20 for tag); ITTAGE uses at most lower 9 bits. Since XiangShan's overriding predictor design, the final prediction result is determined by ITTAGE, which means that matched 9 bits and 32 global history is enough to train a misprediction. Nevertheless, to avoid pipeline bubble caused by BPU redirection which might bring uncertainty to the experiment, We still choose to induce BTB aliasing by matching the lower 29 bits of PC.

5.3 Exploitation Primitives

We explore the possible exploitation primitives by examining the three stages of instruction advancement.

5.3.1 IF. Recent researches have proposed several techniques for exploiting the ICache, such as detecting executable memory [12], tracking cross-process / cross-privillege control flow [15] [3]. These primitives are applicable on both x86 and RISC-V architectures.

5.3.2 ID. Although [10] suggests that exploiting the μop cache can create sufficient instruction execution time differences to serve as a covert channel. To our knowledge, there is no known research that uses the μ op cache as a side channel for attacks. Since the μop cache is independent of the memory subsystem and hence less affected by noise interference, it might be possible to use the Prime + Probe method to more accurately track cross-process / cross-privilege instruction control flow. In addition, the μop cache may also serve as a channel for leaking other types of information. These possibilities require further research and experimentation. These primitives are only available on processor with μop cache, and thus may not be applied on the RISC-V CPU.

5.3.3 EX. The basic exploitation method is similar to that of typical speculative execution. However, Phantom execution means that more instruction combinations could be used to train speculative execution, and some defense mechanisms such as retpoline, which rely on software-level changes on indirect jump instructions, may become ineffective.

6 DOWNFALL

Moghimi[9] introduces the first microarchitectural data sampling exploiting the `gather` instruction, named *Gather Data Sampling*. GDS steal the data leaked by the SIMD register buffer, which is the buffer used for vector instructions like AVX-series. Also, due to the simultaneous multithreading, known as Hyperthreading technology, aggravates the vulnerabilities. The attacker thread shared the same resources with the victim.

6.1 Gather Data Sampling

The `gather` instruction enables user to collect discontinuous data that shares the same base address. It was initially intended for the vector operation, by collecting the scalar data to form a vector. Then `gather` will place them in a vector register. However, some of slot is not intended to be filled. Hence, `gather` requires a mask register offering the mask bits. These bits indicates that, with 1 the data will fill into the corresponding position in vector register, while 0 means the data is discarded and what's already in register remains untouched.

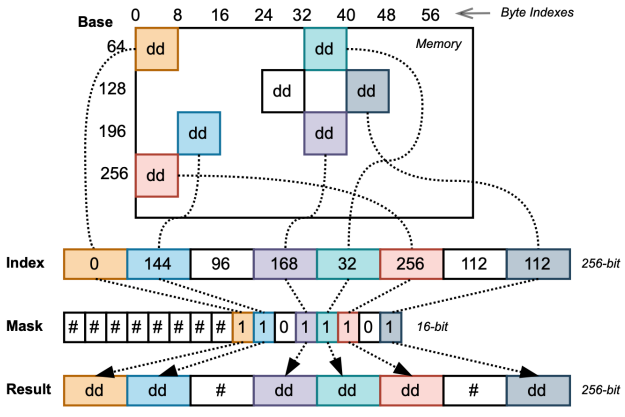

Figure 8: Execution of gather instruction

The author verifies that the processor will

- eliminate unnecessary memory reads for unset mask bits, which must be discarded anyway.

- reuse the data from the same cache line.
- fetch the data in parallel and speculatively and discard results when at least one of the reads has failed.
- preserve the intermediate execution state when interrupt comes, and resume.

These leads to a critical vulnerability. While the original microarchitectural data sampling has a strict assumption, `gather` eliminates it by flooding the SIMD register buffer to reserve the possibly sensitive data in the buffer. After that insurance, attacker can easily perform the MDS attack and steal data from buffer.

6.2 RISC-V Vector Extension

Similar to Intel's AVX extension, RISC-V announced its own vector extension leveraging the SIMD technology. Single Instruction Multiple Data (SIMD) enables the CPU with parallel processing ability, and utilize the computational resource in backend. Meanwhile, the explicit vector length eliminates the cost in branching. RISC-V offer additional 32 vector registers and 7 vector CSR for vector operation. But different from x86, the vector registers are the same width with other registers. RISC-V specifies several instruction and state registers (CSR) to group and control them dynamically. For example, `vlen` sets the vector length and `vtype` can groups the vector registers to form a wider vector register. RVV also provides with rich vector load and store instruction to deal with flexible situation, such as strided and indexed (similar to `gather`).

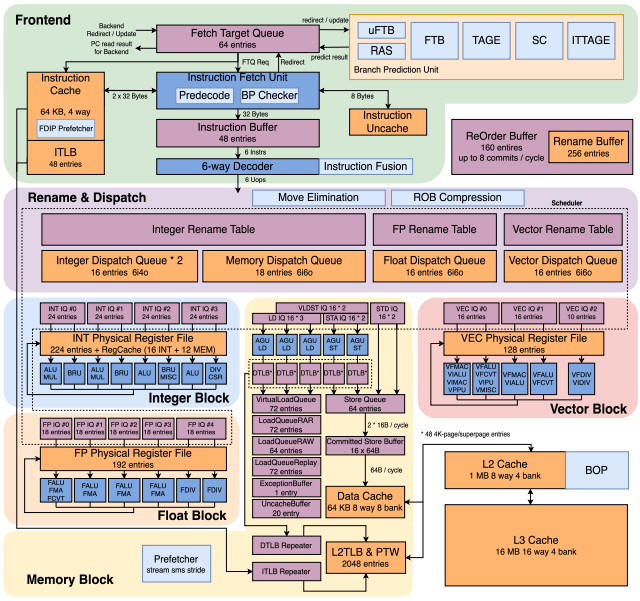

Figure 9: The Kunminghu (3rd) architecture of XiangShan

The XiangShan will decode instruction to μops, which shares similar design with present modern processors, whether it's CISC or RISC. In XiangShan's design, after setting the vector CSR, the vector's decoder will knows how to proceed. The core will decoded the vector computational instruction to several μops and reassemble them in commmit stage. But in the execution stage, all μops will be executed speculatively and out-of-order, same with the scalar instruction. Meanwhile, there's a speculative execution when the computational instruction is following the control instruction (`vtype`), which is often the case. In this situation, the computation will guess the `vtype` since it always comes from immediate number.

Therefore, the vector instruction and other instruction is treated equally at the execution stage – they're all μops. Scalar instruction and vector are executed by the same components, so how can we tell the difference? The answer is we don't, and this enables us to exploit the vector instruction. Similar to Downfall, we can flood the buffers with vector instructions.

6.3 Exploiting RVV Instruction

Spotting similar instruction on XiangShan and on Intel, we can make a reasonable guess that the Downfall attack can be replicating on RISC-V. From Figure 9, the vector buffer is unavailble since vector and scalar has the same execution backend.

However, the shared resource between scalar and vector, such as the store commit buffer and load buffer, magnifies the attack surface, since Downfall is restricted to steal data from SIMD register buffers.

```
fence.i
// increase the transient window
vsetvli t0, %[vl], e64, m1
// Set vector length and element width to 64 bits
vmv.v.x v0, %[mask]
// Move mask to vector register v0
vle32.v v2, (%[indices])
// Load indices into vector register v2
vluxei64.v v1, (%[src]), v2, v0.t
// Load 64-bit elements using indices and mask
vse64.v v1, (%[dst]) Store loaded elements to dst

encode_secret
flush_and_reload
```

Listing 4: Downfall with RVV

In RVV version of Downfall, we firstly increase the transient window and executes a following instruction required to set vector. Next, we perform the GDS and encoded the secret to cache line and finally fetch the secret by a traditional flush and reload.

Our alternative guess on exploiting RVV instruction was to leak inaccessible data by setting the mask bits to 0 and bringing it to cache. By our initial analysis of XiangShan's source code, the guess is possible since the vector load and store instruction will be decoded into several μops that is similar to regular load and store instruction. But the result will be discarded after checking the mask bits and before commit. However, this remains to be tested.

7 EXPERIMENT

We run the experiment on the machines in Table 2

The Intel machine is run under a KVM and host processor, while XiangShan can only run FPGA and simulator. Preparing XiangShan's simulation is pretty tricky, for example, the compilation takes approximately two hours to complete and Verilator runs at

CPU	Generation	Memory
Xeon(R) Silver 4210	Cascade Lake	DDR4 128GB
XiangShan	Nanhu (FPGA)	DDR3 16GB
XiangShan	Kunminghu (Verilator & GEM5)	DDR3 8GB

Table 2: Tested machines

speed of nearly a hundred thousand times slower than the silicon chip. So to verify the attack, we preliminarily ran attacks on the x86-64 processors. Next, we perform the migrated attack on XiangShan. Due to the lmited equipment, we are only able to run Nanhu on FPGA and the latest Kunminghun is available only on simulation by Verilator and GEM5.

7.1 Preliminary

7.1.1 Spectre on x86. We run the unmodified Spectre attack script[8] on Xeon(R) Silver 4210, with cache hit threshold to be 50 cycles. The script was successfully executes and the leakage accuracy reaches nearly 100 percent. Also, we further tested with Spectre to break KASLR and read `/etc/shadow`. The average leakage rate is 2113 bytes per second.

7.1.2 Phantom on x86. Using training method and observation channel mentioned in 5.1.2, we obtain same results on AMD Zen2 as [12], i.e. all combinations of training / victim instructions can leads to phantom execution. However, on AMD Zen3, we could only observe the mispredicted instruction entering ICache, but without subsequent ID and EX signals on instruction types mismatch, which disagrees with the result in [12] where mispredicted instructions can enter the ID stage. This might be because we used different testing metrics.

7.1.3 Downfall on x86. The Downfall attack was already implemented by the author. However, only the testing script and spying scripts can be successfully replicated. The case study to break AES-NI encryption fails to steal any of the keys. The leakage of GDS spy was 6 bytes at average, where the leaked information are pieces of readable string. We can confirm the attack is valid but somehow the case study won't work.

7.2 Exploiting XiangShan

7.2.1 Spectre on XiangShan. Replicating Spectre on XiangShan is tricky due to the following reason

- XiangShan is mostly run on simulator and with bare-metal workload, where libc is unreachable.
- If we try to boot a Linux image, it will cost at least 24 hours to bootup.
- The only applicable solution is to run on FPGA.

As a preliminary test, we measured the cycles elapsed for cache hits and misses. The results shows that the threshold sits around 140 cycles on FPGA with 100MHz clocks and a normal DDR3 RAM. Next we run the migrated RISC-V version of Spectre. The leakage shows a 103 bytes per second and a hundred percent accuracy.

7.2.2 Phantom on XiangShan. We have currently only conducted related experiments in emulation. However, due to the insufficient difference in the values read from the hardware counters, they are not significant enough to serve as indicators for determining whether an instruction has entered a particular stage. We may need to establish more refined observation channels.

7.2.3 Downfall on XiangShan. Playing with RISC-V vector extension requires lots of experience in its details, and the suggested way to apply vector is to invoke the library APIs. However, this violates the principle of our exploitation. Due to the lack of RVV-related knowledge, we are still tweaking the codes and trying to observes the secret brounght to cache.

8 FUTURE WORK

Exploiting the Architecture. Due to the limitation of time and lack of knowledge, the actual attack on the vector extension of RISC-V and phantom execution remains incomplete. We plan to further address the root cause of the malfunctioning attack.

Mitigating the Transient Attack. Hu et al.[5] summarize the possible hardware mitigation against the transient attack. Due to the speculation, software mitigation aims to block the transient window by enabling memory barrier. However, this causes huge performance overhead. Therefore, it's more applicable to implement hardware mitigation, such as the branch shadowing and buffer shadowing. These techniques block the unauthorized access to sensitive data.

9 CONCLUSION

In our project, we investigate the microarchitectural vulnerability of modern processors and focusing on the cache-based side channel attacks. With thorough understanding of the attack surfaces, we preliminarily replicate the Spectre, Phantom and Downfall on x86 machines. Next, we analyze the feasibility in the open-source RISC-V core, XiangShan, and conduct experiment of attacks on it.

From the project, we gain a clear experience and knowledge in the area of microarchitectural security that modern processors are facing with. Being more familiar with those knowledge, we plan to develop hardware mitigation against microarchitectural attacks.

REFERENCES

[1] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2019.

[2] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[3] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. A security risc: Microarchitectural attacks on hardware risc-v cpus. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2321–2338, 2023.

[4] Moein Ghaniyoun, Kristin Barber, Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Teesec: Pre-silicon vulnerability discovery for trusted execution environments. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Guangyuan Hu, Zecheng He, and Ruby B. Lee. Sok: Hardware defenses against speculative execution attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 108–120, 2021.

[6] Hai Jin, Zhuo He, and Weizhong Qiang. Specterminator: Blocking speculative side channels based on instruction classes on risc-v. 20(1), February 2023.

[7] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[8] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. *Commun. ACM*, 63(7):93–101, June 2020.

[9] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[10] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 361–374, 2021.

[11] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[12] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting Decoder-detectable Mispredictions. In *MICRO*, October 2023. Best Paper Award, ETH medal.

[13] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[14] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7267–7284, Anaheim, CA, August 2023. USENIX Association.

[15] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the instruction prefetcher. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7321–7337, Anaheim, CA, August 2023. USENIX Association.