ACA Project
Spectre on XiangShan

Paraula Rey
rey.paraula@mail.polimi.it

Petenzi Roberto
roberto.petenzi@mail.polimi.it

June 27, 2025

POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Abstract

Since their discovery, speculative execution attacks have posed a threat to industrial security, since they are difficult to mitigate without significant loss of performance. The following investigates whether the Spectre attack is doable on XiangShan, an open-source RISC-V processor that aims to become an industry standard in the years to come. The analysis reveals that XiangShan is indeed vulnerable to Spectre and that mitigations introduce critical performance downgrades, which would make the chip less appealing to customers looking for high performance and security.

1 Introduction

Hardware vulnerabilities represent a critical categories of security flaws.
In contrast to software vulnerabilities they are rarely patchable via software updates and can circumvent software security measures: if patches are available they often lead to performance degradation or loss of functionalities. The only reliable way to fix these is to re-design the vulnerable components of the chip, which is a time-consuming and expensive process.
One of the attacks that leverages this class of vulnerabilities is Spectre, discovered by a team at Google Project Zero in 2018.

In this document we analyze the attack itself and try to replicate it on XiangShan, an open-source high-performance processor. This choice is driven by two main factors:

- XiangShan target market are data centers, where confidential and valuable information might be stored.
- The latest version of the chip is still under development using an agile workflow, so mitigations could be introduced before tape-out.

More specifically the goals of our project are:

1. Determining what makes a processor (in our case RISC-V, but the analysis translates to any architecture) vulnerable to Spectre.
2. Analyze XiangShan and its architecture.
3. Replicate the attack on XiangShan and investigate the impact of the simplest (yet effective) mitigation.

This report is divided in the following sections:

- Spectre, which is an overview of the attack and its working mechanisms.
- XiangShan, which describes the relevant characteristics of the processor.
- Primer on cache side channels, which gives an overview of cache side channels and describles the one we used on XiangShan.
- Executing Spectre on XiangShan, which describes how the attack was carried out on XiangShan and the results.
- Statistics, which presents the statistics we collected during our study.
- Mitigation, which describes the implementation of a simple mitigation and its impact on processor's performance.
- Conclusions

The code relative to this project can be found here.

2 Spectre

This section briefly describes the Spectre attack, focusing on the version we executed on XiangShan. More information regarding the Spectre attack, including other variants of it, is available at the Spectre paper[1].

"Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels." [1]. The attack process is as follows:

1. Inject (or locate) a sequence of instruction to leak memory or register content.
2. Trick the processor into speculatively executing this sequence, loading into the cache the target information.
3. Use a cache side channel to extract it.

2.1 Attack versions

The high level description provided above translates practically into different attack versions: the attacker can choose freely how to exploit speculative execution (see Spectre V1 vs Spectre V2 below) or use different micro-architectural side channels. In this report we will focus only on variants which use cache side channels, but other methods are proposed if speculatively executed code is prevented from altering cache state.

Inside the class of Spectre attacks that use cache side channels, two main versions are identified:

- **Spectre V1**: this version exploits branch conditions by tricking the CPU's branch predictor into mis-predicting the direction of a branch and so executing code that shouldn't be run.
- **Spectre V2**: this version exploits indirect branches, using concepts taken from Return Oriented Programming[2]. The attacker chooses a gadget and tricks the branch predictor into jumping to its address.

2.1.1 Spectre V1

This section goes more into details of the Spectre V1 version, which is the one that we run on XiangShan. This decision was driven by two main factors:

- As we will describe later, we will run the attack on a bare metal (with no OS) simulation, so no useful gadgets are available.
- The versions that do not use cache side channels are more complex to implement.

To provide a clearer understanding of how this version operates, consider the following example: the process running the code below accesses two arrays (one smaller, `array1` and one bigger `array2`) and performs a bounds check on `x` as a security measure. It is clear that without the check a malicious user could supply a properly crafted `x` to arbitrarily read memory.

Algorithm 1 c

```c
if(x < array1_size) {
    y = array2[array1[x] * 4096];
}
```

This security measure seems sound if we don't consider the fact that most modern CPU use speculative execution to improve performance. If `array1_size` is not available, the CPU speculatively executes the instructions inside the `if` before checking that `x` is smaller than the array's size, leading to an out-of-bounds access, which will be eventually reverted. This

access, though, will influence the cache and an attacker can use side channels (see Primer on cache side channel) to leak the accessed data.

The attack is executed as follows:

1. The attacker mistrains the BPU (Branch Prediction Unit) to not take the branch (i.e. executing the code inside the `if` block).
2. The attacker uses time consuming operations (in our case FPops) to make `array1_size` not available right away.
3. The attacker uses the cache-side channel to read the accessed data.

2.2 Required components

From the previous analysis we see that the required components for a CPU to be vulnerable to Spectre are:

- **Branch Prediction Unit** (to allow speculative execution)
- **Caches**, with no particular requirements on the number of levels or size. Obviously if the attacker wants to perform a cross-core attack, they would need a shared cache (usually L3). In our case we performed the attack on the L1 cache since we simulated one core.
- **Timers or cycle counters** (to allow the cache side channels).

3 XiangShan

This section analyzes the micro-architecture of XiangShan, in particular on the modules that make the attack possible. It will also present the simulation tools that we used to run the attack.

XiangShan[3] is a super-scalar RISC-V processor, which aims to achieve industrial grade reliability and performance. Its other selling point is that it is a community driven, open source project, developed using an agile workflow, which leads to very fast tape out speeds (2 years for the second generation).

3.1 Micro-architecture and generations

XiangShan has 3 generations:

- V1, Yanqihu
- V2, Nanhu
- V3, Kunminghu

We focused only on V2 and V3 since they are the ones that are actively maintained: XiangShan team's aim is to use the slower and simpler V2 for consumer, large scale products and the faster V3 for industrial and data center operations.

3.1.1 Architectural and micro-architectural differences

Before diving into the micro-architectural modules that allow the attack to be deployed, we will first analyze the differences between the two versions.

Architecturally wise, Kunminghu supports all the latest extensions of the RISC-V ISA[4], including the `zicbom` and `zicboz` extensions. These include Cache Management Operations, which, as we will see later are useful for the cache side channel.

Micro-architecturally wise, for what concerns the attack, the two versions are the same. Indeed the only major micro-architectural difference is the addition of a vector block to boost performance in SIMD-like workflows (such as scientific computing, video processing or cryptography).

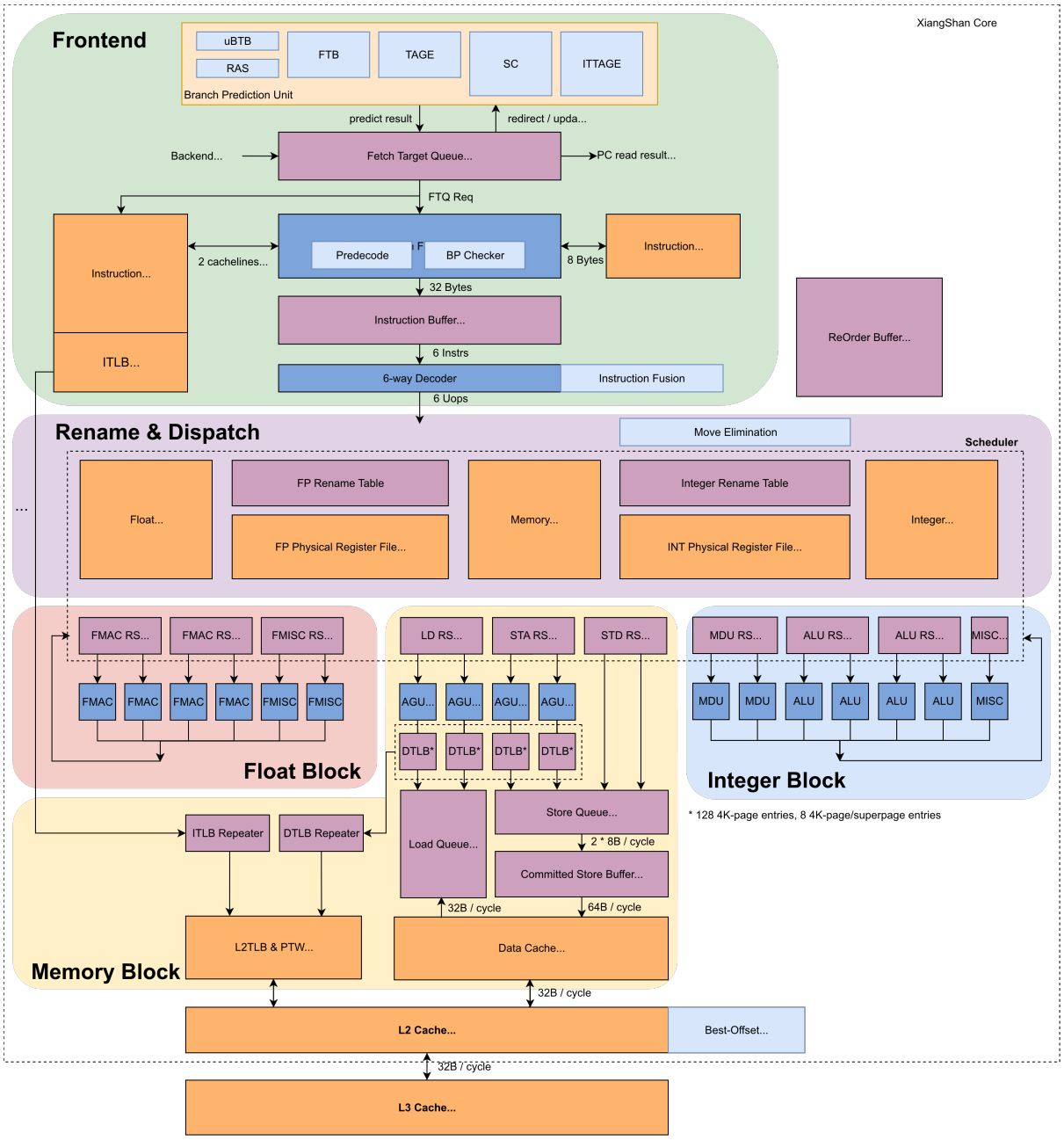

Figure 1: Nanhu micro architecture

As can be seen from the image above, XiangShan has all the prerequisites to be vulnerable to Spectre, since it has both a Branch Predictor Unit and (multiple) cache levels.

Since XiangShan is an open-source project, one can enable/disable or tune each block by editing the configuration file and compiling a new version of the simulator. This makes it very easy to play with different micro-architectural configurations and check the impact of disabling certain modules. It also makes it easy to track changes that might lead to unexpected behavior, as we will see later.

3.1.2 Branch Predictor Unit (BPU)

XiangShan's BPU is a hybrid, multi-stage predictor.

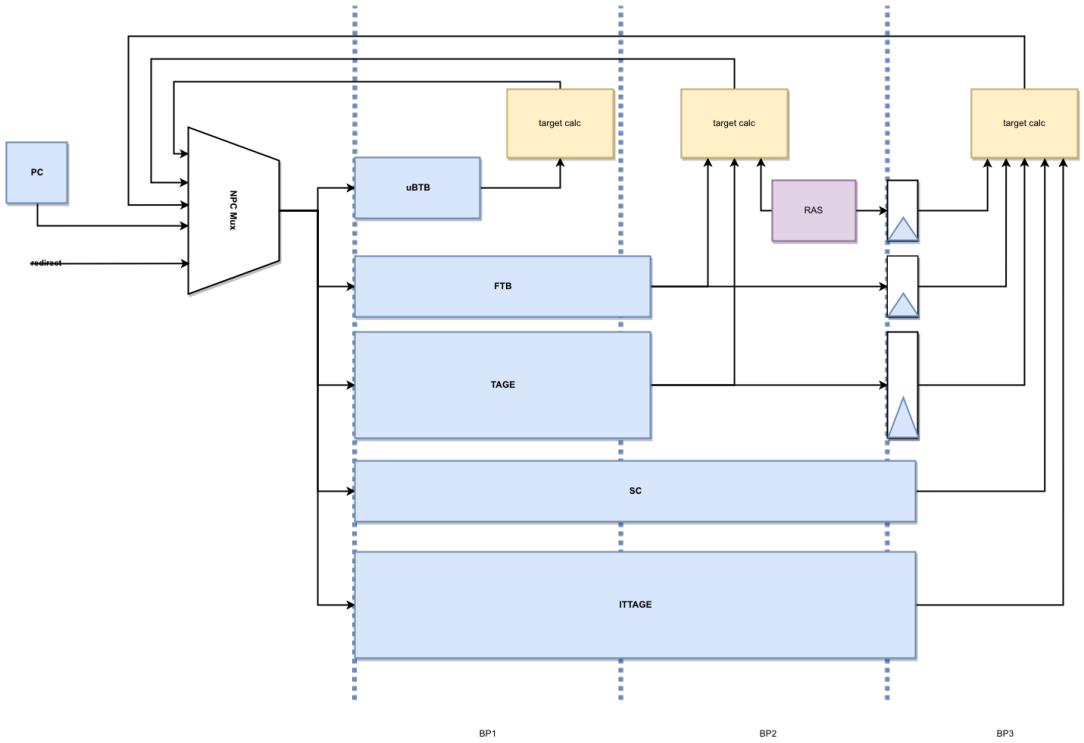

Figure 2: BPU architecture

As seen from the figure above, it uses a fast (1 clock cycle) Next Line Predictor (implemented using a micro BTB, with no tags) and an Accurate Predictor (APD), composed of multiple branch predictors (FTB, TAGE-SC, IITAGE and RAS) with higher latency (2-3 clock cycles).

Predictions from less accurate predictors can be overwritten by more accurate ones to ensure maximum precision.

Each component of the APD is optimized to predict different things:

- FTB: it's the core structure of the APD and defines prediction blocks (chunks of instructions that are analyzed together by the BPU). It can provide information on up to two branches per block, cooperating with other predictors in the unit.
- TAGE-SC: specialized in conditional branches, using exponentially longer history tables (TAGE) and correction logic (SC).
- ITTAGE: specialized in indirect branches, used when FTB or RAS cannot predict the target address accurately.
- RAS: used for function return predictions.

3.1.3 Cache

XiangShan's architecture offers 3 cache levels:

- L1
- L2
- L3, which is shared between cores

The L1 cache is set-associative and uses a pseudo-LRU replacement protocol. It supports error correction protocol such as SECDED ECC to improve reliablilty.

More details can be found on the XiangShan micro-architectural documentation[1]. Since they are not targeted by the attack, we won't go over the details of the L2 and L3 cache.

When deploying the attack we worked with configurations that defined a cache with 8 ways and 64 blocks, each block being 64 Byte long (leading to a 32KB L1 cache size).

3.2 Simulators

Simulation of XiangShan is well supported as it is needed to verify in an Agile fashion implemented functionalities.

There are several simulators available, the most relevant that we analyzed are the following.

3.2.1 NEMU

NEMU[2] is an emulator developed to verify the correctness of the XiangShan processor, the downside found in our case was that it doesn't emulate the cache and so the side-channel cannot be performed.

3.2.2 Verilator

Verilator otherwise is a more general tool that converts an Hardware Description Language to a cycle-accurate behavioral model written in C++, this can be then compiled and executed, fully emulating the entire XiangShan processor.

Its downside is that it is slow, but, being cycle-accurate it can be used to find timing vulnerabilities that can lead to cache side-channels.

4 Primer on cache side-channels

Basically a side-channel attack on the cache allows an attacker to extract sensitive information like secret keys, by measuring access times to specific blocks.

There are several types of side-channels that work on different cache levels and allow to achieve different extraction speeds.

Details can be found at [5].

Our attack exploited the simplest one that is the Flush+Reload documented below

4.1 Flush + Reload

The Flush + Reload attack is divided in two phases:

- The flush phase where the attacker flushes an array that they control, through specific instructions (like `clflush`, `cbo.flush`) or by knowing the eviction policy.
- Wait and reload phase, the attacker waits for a specific amount of time after which checks the access time to each cell. It can be a cache hit or a cache miss and depending on this the information can be extracted.

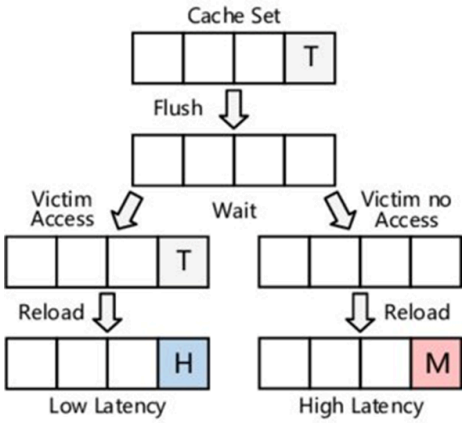

Figure 3: Cache Flush side-channel

[1] https://docs.xiangshan.cc/zh-cn/latest/memory/dcache/dcache/

[2] https://github.com/OpenXiangShan/NEMU

5 Executing Spectre on XiangShan

5.1 XiangShan V2 (Nanhu)

Initially the attack was tested on the stable version of XiangShan that is the V2.

We based our attack on the code that we found in a previous paper [6] that worked on the Berkeley BOOM [7] adapting it to XiangShan architectural parameters like cache settings and access times.

The image below is the output of the attack, the first column (marked by `want(...)`) is the expected value, the other two columns are respectively the highest and the second highest confidence values extracted by the attack.

```
The image is build/spectre-riscv64-xs.bin
want(!) =?= guess 1.(!) 2.(�)
want(") =?= guess 1.(") 2.(|)
want(#) =?= guess 1.(#) 2.(�)
want(T) =?= guess 1.(T) 2.(�)
want(h) =?= guess 1.(h) 2.(�)
want(i) =?= guess 1.(i) 2.(|)
want(s) =?= guess 1.(s) 2.(|)
want(I) =?= guess 1.(I) 2.(�)
want(s) =?= guess 1.(s) 2.(|)
want(T) =?= guess 1.(T) 2.(|)
want(h) =?= guess 1.(h) 2.(�)
want(e) =?= guess 1.(e) 2.(�)
want(B) =?= guess 1.(B) 2.(|)
want(a) =?= guess 1.(a) 2.(|)
want(b) =?= guess 1.(b) 2.(|)
want(y) =?= guess 1.(y) 2.(�)
want(B) =?= guess 1.(B) 2.(|)
want(o) =?= guess 1.(o) 2.(�)
want(o) =?= guess 1.(o) 2.(|)
want(m) =?= guess 1.(m) 2.(|)
want(e) =?= guess 1.(e) 2.(|)
want(r) =?= guess 1.(r) 2.(�)
want(T) =?= guess 1.(T) 2.(|)
want(e) =?= guess 1.(e) 2.(|)
want(s) =?= guess 1.(s) 2.(�)
want(t) =?= guess 1.(t) 2.(|)
```

Figure 4: Output of the Spectre attack

As we can see the highest confidence value matches the expected value, so the attack worked flawlessly and we decided to try it on the latest version.

5.2 XiangShan V3 (Kunmighu)

We tested the same code running it on the `MinimalConfig` available that has same cache parameters as Nanhu.

However due to optimization they introduced it didn't work at first. After debugging we found that, one of the introduced optimizations, added the ability to read Out-of-Order the content of CSR (Control & Status Registers) registers. This caused a wrong read of the counter used to extract the secret from the cache.

To fix this issue we added a `fence` instruction that guaranteed the correct instruction execution order, allowing the side-channel attack to work and actually increasing the confidence of the extracted value. We obtained the same results as the attack on V2 (see Figure 4), with higher extraction speed.

Being the V3 in active development they added the support to newer RISC-V extensions like Cache Management Operations (`zicbom`, `zicboz`) [8] that simplifies the attack by removing the need of knowing and implementing the cache eviction policy to flush the array from the cache.

6 Statistics

6.1 Eviction techniques speed

The first metric we wanted to measure was the impact of eviction techniques. We noticed that changing the eviction function used by the attack led to a noticeable difference in completion time. The three function we decided to compare are the following, which code is available on our GitHub repository[3]:

- `flushCache`: it calculates which cache sets contain the target data, then performs multiple reads from aligned dummy memory to force the cache replacement policy to evict all ways in those sets.
- `evict`: exploits the PLRU algorithm used by XiangShan to evict target data from the cache by accessing memory locations that maps to the same cache set. It uses three nested loops to ensure that all ways are cleared, so it's very complex in terms of execution time.
- `cbo.flush`: this assembly instruction is exposed by the RISC-V ISA and evicts the provided address from cache. Since it is an inline function it provides the highest code readability.

Eviction Function	Cycles (Millions)
`flushCache`	7.02
`evict`	34.25
`cbo.flush`	7.91

Table 1: Speed comparison between different eviction functions

As we see from the figure, the fastest eviction function is the `flushCache` adapted to XiangShan, followed closely by the "native" RISC-V assembly `cbo.flush`. The `evict` is much slower, due to the triple nested loop inside it, that skyrockets the complexity.

6.2 Secret leak speed

After finding out the fastest eviction function we collected some statistics about the secret extraction speed.

Our results have been collected in a Verilog-simulated environment of 1 XiangShan Core getting:

Version	Speed
XiangShan v2	0.98 Byte/Mcycles
XiangShan v3	3.70 Byte/Mcycles

Table 2: Secret extraction speed in simulation environment

We can see that with v3 optimizations also the extraction speed is optimized.

These speeds heavily rely on the simulation environment, benchmarks on real hardware can give for sure better results.

[3] https://github.com/necst/aca25-spectre

7 Mitigations

We explored some possible mitigations like the ones discussed in [9] or [6].

7.1 Disabling the BPU

By having limited time we tested only the most effective one that consists in disabling entirely the Branch Prediction Unit.

To do this we used the `Sbpctl` CSR that XiangShan exposes[4] to disable each branch predictor.

We run on XiangShan v3 the CoreMark [10] benchmark with 2 iterations compiled as a nexus-am[5] bare metal app on the simulator and we got the following results:

BPU	CoreMark score
Enabled	356.63 Iteration/s
Disabled	113.22 Iteration/s

Table 3: CoreMark score with/without BPU

As we can observe this has a big impact on performances but it can be used as a baseline result to compare against more clever mitigations.

8 Conclusion

Our study proved that all evaluated versions of XiangShan are vulnerable to Spectre-V1 attack, even when using the simplest side channel available. In simulation, we were able to extract data with high accuracy, needing just one attack execution to leak the whole secret.

We also implemented the most straightforward mitigation of disabling the entire BPU. This made the attack ineffective, but it introduced a 3x performance degradation, making XiangShan less appealing to customers working with performance intensive tasks where security is a concern, such as data centers.

[4] https://github.com/OpenXiangShan/XiangShan/blob/master/src/main/scala/xiangshan/backend/fu/NewCSR/CSRCustom.scala

[5] https://github.com/OpenXiangShan/nexus-am/

Bibliography

[1] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012, doi: 10.1145/2133375.2133377.

[3] Y. Xu *et al.*, "Towards Developing High Performance RISC-V Processors Using Agile Methodology," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1178–1199. doi: 10.1109/MICRO56248.2022.00080.

[4] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA." [Online]. Available: https://riscv.org/technical/specifications/

[5] F. Rauscher, C. Fiedler, A. Kogler, and D. Gruss, "A Systematic Evaluation of Novel and Existing Cache Side Channels," in *Network and Distributed System Security Symposium (NDSS) 2025*, Feb. 2025. doi: 10.14722/ndss.2025.23253.

[6] A. Gonzalez, "Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204829276

[7] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Jun. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[8] "RISC-V Base Cache Management Operation ISA Extensions." [Online]. Available: https://lists.riscv.org/g/tech-cmo-archived-2022/attachment/865/0/cmobase-v0.5.1.pdf

[9] R. Bălucea and P. Irofti, "Software Mitigation of RISC-V Spectre Attacks," in *Innovative Security Solutions for Information Technology and Communications*, M. Manulis, D. MaimuŢ, and G. Teşeleanu, Eds., Cham: Springer Nature Switzerland, 2024, pp. 51–64.

[10] S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, vol. 6, no. 23, p. 87, 2012, [Online]. Available: https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf