A Genetic Algorithm for a Spectre Attack Agnostic to Branch Predictors

Dorian Bourgeoisat, Laurent Sauvage

A Genetic Algorithm for a Spectre Attack Agnostic to Branch Predictors

Dorian Bourgeoisat
Télécom Paris - Institut Polytechnique de Paris
Paris, France
dorian.bourgeoisat@telecom-paris.fr

Laurent Sauvage
Télécom Paris - Institut Polytechnique de Paris
Paris, France
laurent.sauvage@telecom-paris.fr

ABSTRACT

This paper explores the possibility of using a genetic algorithm to launch a Spectre attack on different branch predictors with the same binary. We focus on RISC-V architectures, specifically SonicBOOM, as they are the most likely to have a wide variety of branch predictors while sharing the same ISA. We show that our genetic algorithm is able to find a training and attack sequence to mount a Spectre Bounds Check Bypass attack on multiple branch predictors.

1 INTRODUCTION

Nowadays, with the rise of information technology, we rely on computer systems to manage financial, medical or military information. Their sensitive nature attracts nefarious actors wanting to abuse them, making cyber threats more and more common, and their consequences, severe.

As some computer programs deal with confidential information, or are critical to health or safety, it is therefore necessary to protect them against attacks seeking to extract this information or preventing their operation. However, multiple users and tasks often have to share the same computing resources to distribute the cost of such systems. In order to avoid information leaks or crashes, isolation measures are implemented, such as Memory Protection or Virtual Memory. These isolation measures have been implemented to prevent code from accessing data they should not access, for example belonging to another task. This mitigates direct secret information extraction, as programs should not be able to access secret information being processed by another program. However, this isolation was found to be imperfect. With cache-based timing attacks, for example, it is possible to observe events correlated with secret data, making it retrievable by timing some process which takes a different amount of time depending on whether or not the event happened. With the Spectre and Meltdown attacks [3, 6–8], it was discovered that not even the hardware can be trusted, as an attacker could influence it to induce observable events depending on the secret, bypassing isolation measures. Though mitigations are available [2, 9, 12], they are costly in terms of performance.

As flaws in these isolation mechanisms are discovered, assumptions made on the security of the underlying abstractions are no longer valid, breaking security guarantees.

To increase performance, multiple techniques are used to execute as much code as possible in the minimum amount of time. One of these techniques enable modern processors to speculatively execute instructions, making use of circuitry that would otherwise be idle. Instead of waiting for the result of a conditional branch, the processor will try to *predict* which branch will be taken and speculatively execute the instructions following it, discarding the results in the case of a misprediction. To maximize the performance gain, the *branch predictor* should predict the branch as accurately as possible, branch predictor designs usually using the branching history of the given branch to derive patterns. Another element, the cache, was introduced to reduce the latency of memory accesses. Caches are small yet fast memories containing data deemed most likely to be used in the near future.

Spectre attacks work by leveraging this speculative execution and using the cache as a covert channel to recover secret data. The attacker will try to manipulate the branch predictor to speculatively execute an instruction that will change the state of the cache, enabling a timing attack.

As Spectre attacks rely on speculative execution, they are sensitive to changes in the branch prediction algorithms.

RISC-V is an open ISA with a royalty-free specification. Thanks to this openness, it is becoming increasingly widespread in small embedded systems, with Western Digital having a RISC-V controller for their drives, and other vendors building their own microarchitecures. With this openness and use in many different specialized applications, many microarchitecures with completely different implementations of the ISA are becoming available to run software on.

For example, while Spectre attacks were replicated on BOOMv2 [5], it is not replicable as-is on BOOMv3 as the branch predictor was changed from a gshare branch predictor to a TAGE predictor with a loop predictor [13, 18].

As RISC-V is an open ISA with many implementations, it is more likely to have a large diversity in branch predictors, hindering a widespread attack on all RISC-V implementations.

While such diversity might prevent the spread of a targeted attack to other microarchitectures through the same binary, an attack treating branch predictors as black boxes would enable attackers to target a wider variety of microarchitectures with only one binary.

In this paper, we show how a simple genetic algorithm can be used to find how to mount a Spectre attack on different branch predictors with the same binary, being completely agnostic to the inner workings of the branch predictors. In Section 2, we present the basics of Spectre attacks and the branch predictors targeted by them. In Section 3, we reframe the Spectre Bounds Check Bypass attack in terms of a sequence of arguments which can be evaluated to see whether the attack yields accurate results. We then propose in Section 4 a simple genetic algorithm to find a sequence that, with the Spectre access time measuring setup, yields the most accurate recovery of a secret. The results presented in Section 6 show the method's effectiveness by making the algorithm exhibit a training-and-attack sequence for different branch predictors, using the same binary, on the BOOMv3 RISC-V microarchitecture.

2 PRELIMINARIES

2.1 Spectre v1

Spectre is a class of micro-architectural attacks relying on a misprediction triggering a change in the cache state observable by the attacker. Multiple variants of Spectre exist, some targeting indirect branch prediction, others the return stack buffer [7]. The specific variant we are targeting is the Bounds Check Bypass attack, often called Spectre v1 [6]. This attacker needs to target a code gadget in the attacked code that is vulnerable to this flaw.

This gadget needs to be of this form:

(1) A conditional branch on a value `x` that can be controlled by the attacker.
(2) In the branch, a load of a value `y` from a location dependent on the attacker-controlled `x`.
(3) Following that, another load from a location dependent on `y`.

The attack consists of:

(1) Executing the gadget with values such that the branch predictor's next prediction on the branch is to take it.
(2) Executing the gadget with a specific value as an argument such that `y` is the secret.
(3) The latency of an access to the location dependent on `y` being different due to the change in cache state induced by the previous step, the attacker can infer `y`, therefore the secret.

For example, considering the following gadget:

```
if (x < array1_size) {
  _ = array2[array1[x]];
}
```

By executing this gadget with a controlled `x` value, the attacker can train the branch predictor to predict that the branch will be taken. The attacker then executes the gadget with an `x` value such that `array1 + x` is the address of a secret value, even if the condition is false. The branch predictor will predict that the branch will be taken, and the processor will execute the subsequent instructions speculatively. Before the resolution of the branch, the processor will get the result of the load at `array1 + x`, the secret, and execute a load of the value at `array2 + secret`. The execution of this last load will bring in the cache the value at `array2 + secret`. By timing the latency of the `array2` array, the secret value can be deduced. This can be done by methods such as a Prime+Probe [11] or Flush+Reload [16]. Other variants exist, using other types of branching and therefore the kinds of branch predictors used to predict those branches, such as return target buffers or indirect branch predictors. However, they do not enter the scope of this paper.

2.2 Targets of the attack

Various micro-architectures are susceptible to such attacks. They may use all kinds of ISAs, such as x86, but such attacks were also replicated on microarchitectures using different ISAs, such as ARM, or even RISC-V [5]. This is because the Spectre attacks does not rely on a specific ISA, but on the use of speculative execution. This technique is implemented in many microarchitectures intended for relatively high performance applications. Indeed, this technique improves the pipeline efficiency so much that the performance improvements cannot be ignored, therefore many microarchitectures are vulnerable to Spectre attacks, such as BOOM, a RISC-V microarchitecture aiming at performance. Yet while the attacks rely on the use of this technique, it is worth noting that not only the presence of speculative execution is necessary, but the branch predictor needs to be trained aptly to correctly execute the attack, and there is a huge diversity in branch predictors.

Apart from static branch predictors, which can be trivially attacked without training, or simply cannot, depending on how the program was compiled, most branch predictors rely on the previous executions to make an informed guess.

For example, the `gshare` branch predictor uses a global history of taken/not taken branches and the branch address to select a 2-bit saturating counter. This means that by calling the code gadget enough times, the branch predictor will, for the given address and global history, have saturated the counter, making it reliably mispredict. However the `TAGE-L` branch predictor not only uses a global history to select a backing branch predictor, but also a loop predictor [13]. A loop predictor prevents a repeated attack with a fixed-length training sequence, as with time, the loop predictor will correctly predict when the branch will not be taken. Also, it is important to keep in mind that these predictors can be have multiple versions by varying the parameters such as branch history length, also using a local history, etc... With this diversity, while the general behaviour of most branch predictors is understood, an attack relying on the specific behaviour of a branch predictor is open to a simple change in its implementation causing the attack to fail.

3 PROBLEM DEFINITION

The attacker, while writing their attack, would like to have the best attack sequence for the microarchitecture their attack is running on, and have their attack work on as many microarchitectures as possible as it could be an unknown to the attacker. Their problem is therefore: How to write an attack that targets accurately as many microarchitectures as possible?

To deal with the variety of possible branch predictors, an ideal attack would be able to adapt without prior knowledge of branch predictor behaviour, so that it could run on any vulnerable microarchitecture.

Previous work focuses on discovering new attacks or rediscovering existing ones, through reverse engineering [4], or through fuzzing [10, 15]. While these methods are useful to discover new variants or to check that a countermeasure works, they take either a lot of human time and require the adaptation to be done before compiling the attack, thus limiting the number of branch predictors the attack targets to those the programmer implemented support for, or a lot of time to run because of the breadth of the search, and therefore are not suited for use in an attack where the attacker could be discovered and stopped.

We try to address these issues by incorporating knowledge of the basic building blocks of the Spectre V1 attack into an automated method, while treating the branch predictor's behaviour as a black box.

Using Spectre V1, the attacker is trying to find a way to make the branch predictor mispredict on a branch instruction such that the misprediction could be exploited to leak data. In existing Spectre attacks, it is achieved though a training sequence, where the target gadget is called with x values such that the branch predictor "learns" to predict a "taken" branch, then when the predictor is trained, the gadget is called with the chosen x value to attack.

The entire attack sequence, from the attacker's point of view, can be therefore summarized as the list of calls to the target functions, which can be further reduced to the list of arguments used.

Given the characteristics of the Spectre v1 attack, we then define a solution as a sequence of x_i values used to train the branch predictor, with a flag to indicate when to use the attacking x value.

A solution to the problem should have the following characteristics, in decreasing order of importance:

- An attack using the call sequence should retrieve the secret data as accurately as possible.
- It should be as fast as possible.

Other characteristics might also be desirable for an attack but fall outside the scope of this paper.

We recognise here an optimisation problem: for the architecture the code is running on, which solution makes the attack work the most accurately, and the fastest?

4 ALGORITHM

4.1 Overview

Genetic algorithms are simple but effective optimisation algorithms. They use principles of natural selection to select the best solutions among candidates, and create new candidates from them.

A solution is a set of parameters that can be evaluated using a fitness function determining their use as a base for the next generation. New candidates are created by *mutating* the parameters and *crossing over* with other solutions.

In addition, some techniques, such as *elitism*, are added to this simple algorithm to avoid undesirable behaviours. *Elitism* consists of keeping the best solutions of the previous generation in the next generation, in order to avoid throwing away good parameters with little hope of getting them back in future generations.

We decide to try a genetic algorithm because the format of a solution to the problem closely resembles a genome: a sequence of values the interpretation of which gives a result whose fitness can be measured, which, in the case of a Spectre V1 attack, a sequence of arguments to call the gadget with whose effectiveness to extract the secret is evaluated.

Our genetic algorithm is composed of the following steps:

1. **Initialise the population** with the null solution, which corresponds to repeatedly calling the gadget with 0 as the attacker-controlled value.
2. **Evaluate the fitness** (see 4.2) of each solution by using them in an attack against a dummy target function using the gadget.
3. **Keep the best solution** of the previous generation.
4. **Generate new candidates** (see 4.3) by *mutating* and *crossing over* the best solutions using the following substeps:
   - 4.1 Randomly swap the parameters of the 2 best solutions (*crossing over*).
   - 4.2 Generate the other candidates by mutating one of the 2 solutions and randomly swapping two consecutive parameters. This means that we randomly modify arguments in the sequence of calls and randomly swap them in the sequence order.
5. Repeat steps 2 to 5 until the maximum number of generations is reached.

4.2 Fitness Function

To have a solution with the desired characteristics, we designed the fitness function to take into account the following parameters:

- Time taken to execute the attack, in the number of executions of the gadget.
- The ability to recover the secret value as the first guess.
- The ability to recover the secret value as the second guess.

As a multi-objective optimization problem, the importance of the objectives are parametrized in the fitness formula.

The formula chosen to evaluate the fitness of a solution is:

$$F = R_f * C_f + R_s * C_s + T + B$$

Where:

R_f is the number of bytes of the secret found as first guesses.
R_s is the number of bytes of the secret found as second guesses.
C_f is the score coefficient of first guess recovery.
C_s is the score coefficient of second guess recovery.
T is a time score based on the number of calls to the gadget.
B is a bonus score for finding close to the entire secret.

With

$$B = C_b * (\frac{R_f + R_s}{S})^2$$

S being the secret's size and C_b being the bonus coefficient.

$$T = (N_b - N_s) * C_t$$

N_b being a reference number of calls to the gadget, N_s being the measured number of calls, and C_t being a coefficient.

4.3 Generation of new candidates

Contrary to most genetic algorithms, the genes we use, the arguments to be used in sequence with the gadget, are mostly irrelevant on their own, as they cannot have their fitness evaluated independently. This behaviour is caused by the fact only the way to evaluate the fitness of a set of genes is to try an attack and that most of the problem is "when" to attack, and is likely not affected by the value of x_i as long as it verifies the condition. However, it is important to keep this ability to have different values in case a branch predictor is able to use this information to correctly predict the branch.

This is why it is important to allow genes to move in the sequence. Moving genes allow not only for a more diverse population, but also permit faster convergence in case the optimal solution needs a specific number of "taken" calls to be made to successfully attack.

We generate the next generation by first swapping the genes of the two best solutions with a probability α, then we generate the

other solutions by applying mutations or swaps to one of the two solutions thus generated.

Two consecutive genes are swapped with a probability β. This is to allow the genes to move in the sequence. Then we can apply a mutation to the gene with a probability γ. The mutation is chosen with a uniform distribution in the interval $[a, b]$.

5 TESTING METHOLOGY

To test the algorithm we used a dummy function susceptible to a Spectre Bounds Check Bypass attack, a randomly-generated secret, and a function using a solution to attack and get the secret from the side channel. We then implemented the genetic algorithm, and wrapped it in a loop to test it several times. To prevent previous attempts from interfering with the current one, we add random calls to the dummy function so the branch predictor cannot keep a behaviour induced by previous candidates.

Both the genetic algorithm and the attacking code run on the attacked processor.

In order to complete the different tests in a timely manner, we decided to forgo simulating the BOOM RISC-V microarchitecture and instead use a board with an FPGA, and synthesized the BOOM core. The board we used is a Diligent Nexys Video.

Due to resource constraints, the available memory was taken into account in the parameters of the program.

We executed the program using, for RISC V targets, BOOMv3 as a base, and configuring it to use different branch predictors. The cores thus generated were then synthsized and put on a FPGA.

The following branch predictors were tested:

1. `TAGE-L`, natively used by BOOMv3
2. `Alpha21264`, also available in the BOOM codebase

Their configuration was kept at their default values in the BOOM codebase, which, to the time of writing, are as described in Table 1.

Table 1: Branch predictor configurations

Parameter	TAGE-L	Alpha21264
Max Meta Length	120	64
Global History Length	64	32
Local History Length	1	32
Local History Sets	0	128

5.1 Parameters

We tested the algorithm using the fitness parameters described in Table 2 and the algorithm constants described in Table 3. The secret size was chosen to be 10 bytes arbitrarily. The rationale behind the choice of small genome size, generation size and number of generations is to keep the execution time low enough, taking the performance of the microarchitecture into account. N_b was set corresponding to the order of magnitude of calls to the gadget reached after a few iterations. C_t was set to $\frac{1}{64}$ as a scaling factor. C_f and C_s were set to 10 and 5 respectively, to give more importance to first guess recovery, and C_b was set to 100 in order to double the score if the entire secret is recovered.

Table 2: Parameters used for the fitness function

Parameter	Value
C_f	10
C_s	5
C_b	100
C_t	$\frac{1}{64}$
N_b	1000

Table 3: Parameters of the genetic algorithm

Parameter	Value
Genome size	1024
Generation size	8
Number of generations	32
α	$\frac{1}{2}$
β	$\frac{1}{8}$
γ	$\frac{1}{16}$
a	-4
b	3

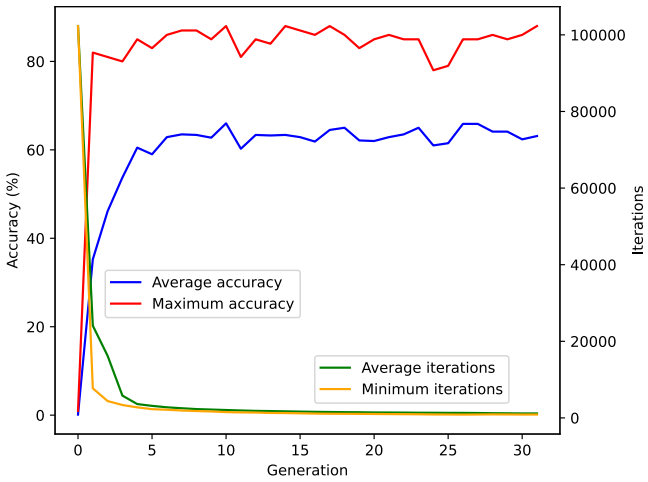

Figure 1: Performance of the generated solutions on the TAGE-L branch predictor by generation

6 RESULTS

On the TAGE-L branch predictor, the algorithm managed to converge very quickly towards an acceptable solution, as 80% accuracy was hit at the second generation as shown in Figure 1. We can also see the the algorithm struggles to have an accuracy better than 85%. While it is certain that an attack specifically written for this branch predictor could achieve a near-100% accuracy, and our algorithm is unable to generate a solution with better that 90%, our algorithm treated the branch prediction behaviour as a black blox and thus did not rely on the programmer targeting the specific microarchitecture. On Figure 2 we can see that the algorithm is able to generate solutions that are far better at extracting the secret than

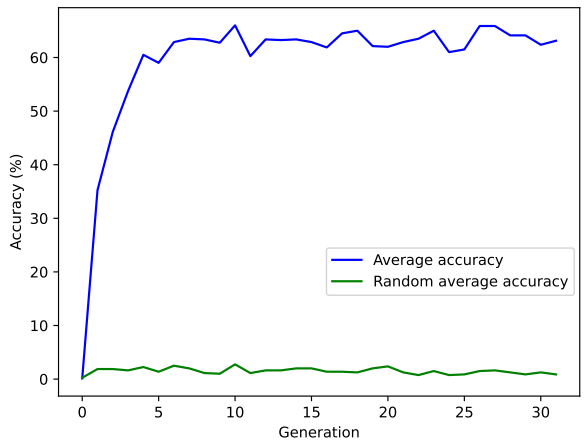

Figure 2: Comparison of the accuracy of the generated solutions on the TAGE-L branch predictor by generation versus random solutions

random solutions. Random solutions were generated as a sequence of random 32 bits integers sampled from a uniform distribution.

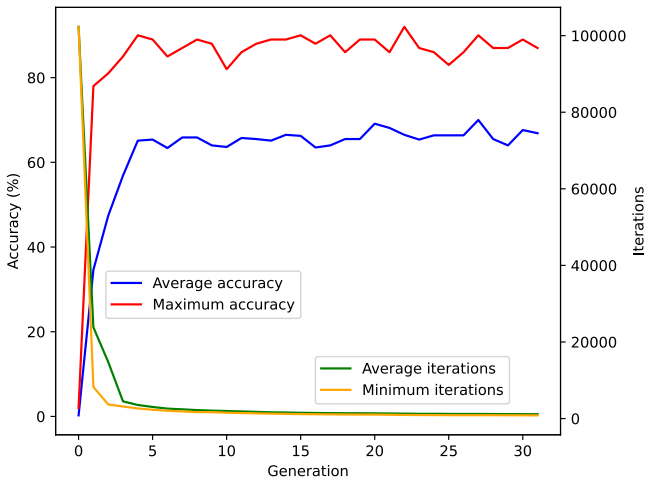

Figure 3: Performance of the generated solutions on the Alpha21264 branch predictor by generation

We have very similar results to the TAGE-L branch predictor on the Alpha21264 branch predictor, but the maximum accuracy of a generation can get a bit higher, sometimes beating the 90%. The average accuracy follows the same trend with a slight increase from the TAGE-L results. This can be attributed to the the increased complexity of the TAGE-L branch predictor, meaning that the behaviour of the Alpha21264 branch predictor is more predictable.

7 CONCLUSION

The simple genetic algorithm we showed in this article is able to find a training and attacking sequence to successfully extract a secret using a Spectre Bounds Check Bypass attack on multiple branch predictors, being agnostic to their inner workings. It is able to achieve this in a relatively small number of generations, and its accuracy, though not perfect, makes a Spectre attack against a wider range of microarchitectures feasible in a single binary. With this algorithm, the training and attacking code doesn't have to be re-implemented for each microarchitecture, and therefore, less microarchitecture specific code has to be written for an attack. This means that to defend against the Spectre class of attacks, it is not possible to rely on branch predictor diversity to avoid a widespread attack, meaning that dedicated countermeasures such as GhostMinion [2] have to be implemented.

8 FUTURE WORK

A simple genetic algorithm provided good results, but other algorithms might yield better and more accurate results, such as Artificial Neural Networks, or more complex genetic algorithms. We also see Q-learning algorithms as possibly yielding better results with faster convergence. Moreover, this method could be tested on other RISC-V microarchitectures or other ISAs to evaluate its effectiveness on other branch predictors.

Our work focused on branch prediction diversity, but does not address the diversity in cache configurations. As RISC-V does not include an instruction to invalidate a cache line, clearing the cache for the attack necessitates prior knowledge of the cache configuration. Moreover, our work did not focus on avoiding prefetch interference, nor on how to extract the secret from the timing data. A truly generic attack would require to also implement a method to dynamically infer the cache configuration, avoid prefetch interference in retrieving the information and an agnostic way to infer the secret. Fortunately, the first problem already has a solution as it was needed to optimise programs for architectures with poorly documented memory hierarchies [1, 17]. The second problem already was addressed in the original Spectre paper [6] with a randomised read order as recent x86 processors implement a prefetch, and the third problem might be able to be solved by a Machine Learning algorithm as a demonstration of such a method was tried [14]. A binary implementing those solutions should be generic enough to be able to extract secrets on all microarchitectures sharing the same ISA if they are susceptible to a Bounds Check Bypass attack.

REFERENCES

[1] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, pp. 65–74. [Online]. Available: https://doi.org/10.1109/RTAS.2013.6531080

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

[3] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 785–800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[4] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *SIGPLAN Not.*, vol. 53, no. 2, p. 693–707, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173204

[5] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović, "Replicating and mitigating Spectre Attacks on a open source RISC-V microarchitecture," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019. [Online]. Available: https://carrv.github.io/2019/papers/carrv2019_paper_5.pdf

[6] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1–19.

[7] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Conference on Offensive Technologies*, ser. WOOT'18. USA: USENIX Association, 2018, p. 3.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[9] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*, J. Szefer, W. Shi, and R. B. Lee, Eds. ACM, 2018, pp. 1:1–1:8. [Online]. Available: https://doi.org/10.1145/3214292.3214293

[10] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, March 2022. [Online]. Available: https://www.microsoft.com/en-us/research/publication/revizor-testing-black-box-cpus-against-speculation-contracts/

[11] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA 2006*, ser. LNCS, D. Pointcheval, Ed., vol. 3860. Springer, Heidelberg, Feb. 2006, pp. 1–20.

[12] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 73–86. [Online]. Available: https://doi.org/10.1145/3352460.3358314

[13] A. Seznec, "A new case for the TAGE branch predictor," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, C. Galuzzi, L. Carro, A. Moshovos, and M. Prvulovic, Eds. ACM, 2011, pp. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[14] J. Snell, "Deep spectre," https://github.com/asm/deep_spectre, accessed: 2022-11-21.

[15] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1415–1432. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/weber

[16] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security 2014*, K. Fu and J. Jung, Eds. USENIX Association, Aug. 2014, pp. 719–732.

[17] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada*, D. L. Eager, C. L. Williamson, S. C. Borst, and J. C. S. Lui, Eds. ACM, 2005, pp. 181–192. [Online]. Available: https://doi.org/10.1145/1064212.1064233

[18] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanović, "SonicBOOM: The 3rd generation berkeley out-of-order machine," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020. [Online]. Available: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf