Recovering Information on the CVA6 RISC-V CPU with a Baremetal Micro-Architectural Covert Channel

Valentin Martinoli[1,2], Yannick Teglia[1], Bouagoun Abdellah[1], Régis Leveugle[2]
[1] Cybersecurity Hardware lab, Thales DIS, La Ciotat, France, valentin.martinoli@external.thalesgroup.com, yannick.teglia@thalesgroup.com, abdellahbouagoun@gmail.com
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP* , TIMA, 38000 Grenoble, France, firstname.name@univ-grenoble-alpes.fr
*Institute of Engineering Univ. Grenoble Alpes

2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS) | 978-1-6654-7355-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/IOLTS56730.2022.9897297

***Abstract*—In this work, we study the micro architectural vulnerabilities of the open-source RISC-V CPU named CVA6. We build a realistic scenario for extracting information and propose an analysis on how to reduce the impact of noise on the attack, while staying as close as possible to hardware level through baremetal simulations.**

Keywords— hardware security, micro-architecture, covert channel, timing side-channels, RISC-V, CVA6, Prime+Probe.

I. Introduction

Micro architectural attacks take advantage of optimization mechanisms inside recent CPUs to cause information leakages. Competitive access to limited and shared hardware resources is the root cause of micro architectural covert channels. We use this behavior to implement a Prime + Probe micro architectural covert channel, and we simulate extraction of secret information through the L1 data cache of the CVA6, a 64-bit open-source application class RISC-V processor. We show the different challenges that need to be taken up to adapt a well-known micro architectural covert channel to a specific attack scenario. Moreover, we show the implications that adding an Operating System might have on the said attack by adding a scheduling mechanism and generating noise. This enables us to study the impact of scheduling and multiprocessing on our micro architectural timing covert channel. We conclude that tailoring a micro architectural attack that has already been applied to a different attack scenario is far from being instantaneous and easy, and requires deep knowledge on both the target and the attack itself. We make the following contributions:

1) Showing a baremetal implementation of the Prime+Probe micro architectural attack on the CVA6 core and discuss the challenges about its potential improvements,

2) Analyzing the effectiveness of the implemented attack in a realistic scenario facing noise and scheduling processes,

3) Discussing the possibilities of adapting a specific attack to a given software and hardware combination to make it as powerful as possible.

After presenting more details on the context in section II, section III explains the baremetal implementation of the attack. Section IV discusses the impact of additional noise that may be due e.g., to the use of an OS, before the conclusions.

II. Background

A. Threat Model

The applicative scenario we have chosen is based on a victim application that runs computations implying a secret value, and is isolated from other processes. We suppose that this application contains either a Trojan, or a buggy implementation that causes leakages at the micro architectural level. We consider an attacker that is aware of this leakage, and willing to recover the secret used by the victim process. Even if the attacker and the victim are logically isolated, the former will use the micro architectural information leakage to recover the secret information.

As for the technical setup, let us consider first a mono-core and mono-thread baremetal scenario. The victim is run on a RISC-V core and its implementation prohibits it from forwarding directly the secret data to another application.

Assume the victim program is compromised by a Trojan that is actively trying to leak the secret information from the victim application. The hypothesis of the victim being compromised is legitimate as shown by the recent Ripple20 series of vulnerabilities [1]. The possibility to hide an adversary code was discovered in a widely deployed TCP/IP library used in several domains.

The attacker itself is contained in a second baremetal application. He time-shares the core with the victim application. This attacker runs a spy program that tries to recover the information leaked by the Trojan. The threat model is summarized on Fig. 1.

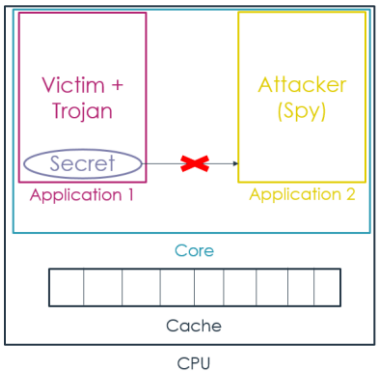

Fig. 1. Chosen threat model for the baremetal attack's context

B. Micro architectural timing covert channels

Traditional covert-channels are well known as making possible to transfer information between processes that should not be able to share data according to the running security policy. Recently, the development of attacks such as Foreshadow [2] and the MDS (Micro architectural Data Sampling) attacks [3-5] led to a new type of covert channel that can transfer information from micro architectural structures to the architectural world that can be observed. The most common type of micro architectural covert channel in the recent attacks is the access-driven cache-based mechanism. Caches are shared among different processes and

are the "closest" micro architectural structure making it the ideal target for an attacker. Many variants of attacks have been published such as Prime + Probe [6], Flush + Reload [7], Flush + Flush [8], Prime & Abort [9].

While the cache's existence is oriented towards performance, it can be exploited in a malicious way as a side channel and across security boundary between processes. All these attacks' variants rely on the capability for an attacker to observe whether a given cache line has been evicted. This is used to infer information about the secret to be extracted. This category of covert-channels attacks is very similar to the time-driven attacks as the source of information leakage comes from timing considerations. However, access-driven cache-based attacks study the cache's behavior at a finer level of granularity.

C. Prime+Probe

The Prime+Probe technique can be divided into five successive steps:

1) The spy places the exploited hardware element such as the L1 data cache in a state where the attacker knows its content, filling it with its own data. After this prime phase, the spy further accessing its own data will result in a faster response compared to a potential eviction.

2) The spy's execution stops and the victim containing the Trojan is then executed.

3) During the victim's execution, the Trojan will also be computed and part of the L1 data cache's addresses will be used. It causes the eviction of some of the spy's data. The Trojan then encodes the secret information inside the data cache. This encoding can take many forms, but we will here take a simple example for clarification purposes. Let us consider a secret value named "s". The Trojan will encode "s" inside the cache by replacing exactly "s" cache lines within the L1 data cache.

4) The victim and Trojan executions end. The spy now probes the time it takes to access the cache lines by reading its own data. As the Trojan replaced some of them, these data will have a longer access time than others, due to cache misses.

5) The spy knows the total access time when all of its data are still in the cache. The previous step gives the total access time when data has been removed from the cache because of the Trojan's action. The difference between these two measurements indicates how many lines have been modified in the cache and (hopefully) the secret value "s".

III. Recovering Secret Information in Baremetal Using the Prime+Probe Covert Channel

A. Experimental setup

In order to assess the level of vulnerability of the CVA6 core itself, we chose to implement a Prime+Probe attack in a baremetal context. The goal of this study was also to pinpoint the exact micro architectural primitives inside the CVA6 for triggering such an attack.

To carry out our experimentations, we chose the open-source toolchain and processor description provided by the Open-Hardware group [10]. CVA6 was used in the default version, with a 32 KB 8-way associative data cache configuration and 16-byte lines. It is in a write-through no-write allocate configuration [11]. The toolchain comes with several simulation and development tools, including the cycle accurate simulator Verilator [12]. We slightly adapted a test suite from the toolchain to include our attack codes, written in C. We also used the Spike Instruction Set Architecture [13] simulator for coherency checks.

B. Our implementation of the Prime+Probe covert channel

Our implementation of the attack contains some inline assembly commands adapted to the baremetal setup. The attacker and the victim are in this context two different functions running concurrently, and being "scheduled artificially" by ourselves inside the code itself as shown on Fig. 2.

```
Prime+Probe_artificial_scheduling(void)
1   Run priming_phase()
2   Run victim_process(secret_value)
3   Run probing_phase()
4   Return(secret_value)
```

Fig. 2. Artificial scheduling's pseudocode

We chose to work on cache sets rather than on cache lines. The CVA6's eviction policy is pseudo-random [10] and this characteristic motivated our choice. Even if it reduces the amount of values that the attack can extract, it also makes it easier as it reduces its granularity, and the impact of the eviction policy. We used while loops affecting values to data structures corresponding to the size of a single zone inside the cache to proceed to the priming phase, from set 0 to set 255. The data cache is totally filled with attacker data after this step.

Then the victim and the Trojan are run. Our victim simply consisted in an addition operation whose result was our secret value. As for the Trojan, we chose a basic encoding strategy, for simplification purposes. Therefore, the Trojan encodes the secret value *s* by completely filling sets of data inside the cache. The Trojan will fill the sets numbered from 0 to *s*-1. This means we can theoretically extract values up to 255.

The spy code is then run again and it measures the access time to its data. We used the privileged machine mode instruction *mcycle* that accesses a control register (CSR) storing the CPU's clock cycle to measure the amount of cycles it takes to access a given cache set. By calibrating our threshold through several tests, we were able to distinguish cache hits from cache misses using this instruction, and thus to recover the secret value in the spy code.

Considering the specific configuration and design of the CVA6's data cache presented earlier (see section III. A.), we had to adapt the Prime+Probe general attack pattern. The main modification is related to the data cache's filling policy: write-through/no-write-allocate. This causes the data to be brought into the cache only on read misses. Therefore, for the priming step, we need to ensure that the attacker's data we want to fill the cache with is both written and read by the spy code. Failure to comply with the previous statement would cause the data to be loaded only in main memory. For this reason, our attack code's "priming_phase()" shown on Fig. 2 differs from the classical implementation. Indeed, each data is both written and read again to comply with the CVA6's specific cache policy and ensure we are causing a read miss.

Moreover, according to the CVA6's design, the data cache is composed of two separate physical SRAMs called "banks". This implies that every cache line is physically split into two parts. This specific design choice has an impact on the implemented attack. For the priming phase to work optimally, the attacker needs the cache to be completely filled, therefore taking into consideration such design specificities is mandatory. Fig. 3. gives a representation of the CVA6's data cache structure and of the cache's filling according to its design.

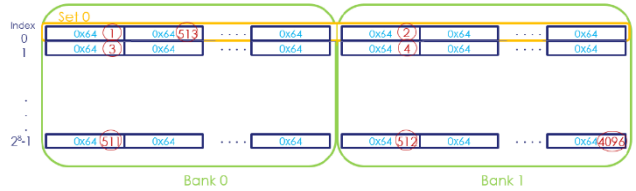

Fig. 3. Representation of the CVA6's data cache structure and filling policy (from 1 to 4096). The value 0x64 has been chosen as the attacker controlled value in this example.

This specific structure has also implications on the attack itself. Indeed, when access to a specific cache line is requested, both blocks are loaded, even if only one is required resulting in 128 bits (the size of a cache line) being handled. For simplification purposes, and to avoid redundancy, we chose to focus on the content of the bank 0 in our experimentations. As a result, the evictions caused by the Trojan's activity are only located inside the bank 0. Fig. 4. shows the positions of the cache replacements obtained with our implementation of the Trojan.

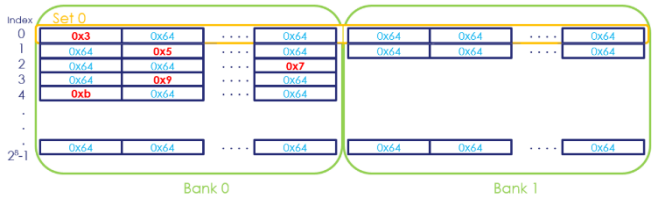

Fig. 4. Representation of the evictions caused by the implemented Trojan's activity inside the CVA6's data cache

These observations further enforce the choice of working on cache sets instead of cache lines, as it is more in agreement with the cache's filling policy: it is easier to predict in which cache set the eviction will occur than to choose or predict precisely which cache line will be evicted, even if it is not impossible (the cache line replacement policy is pseudo-random and highly predictable as coming from a simple LFSR).

Our main idea was to propose an encoding technique that would resist to unwanted evictions more than considering the classical increase in total cache access latency. The initial idea was to be able to recover the secret information even if we had unplanned perturbations caused by another process running. We also chose to consider the worst-case scenario: a process that can cause evictions between the priming and the probing phase, thus altering the cache's state before the information is extracted.

We expected that this encoding technique would be more resilient than the classical attack considering the increase in the total latency as the source of information. Indeed, such a worst-case perturbation would probably cause the classical attack to fail in a mono-core mono-thread scenario, as the total latency would be increased by the processing time of the unwanted code thus modifying the transmitted value. This expected noise resistance will be discussed further in a later section below.

C. *Results obtained*

Our proposed implementation, even simple, could extract secret values between 0 and 50 with more than 90% of success rate. For higher values, the success rate decreases slightly: between 51 and 130, the success rate is approximately 85%; and for values between 131 and 195, the success rate is around 70%. However, these values are highly code and hardware dependent and change from one software implementation to another and from one cache configuration to another. We could not extract values higher than 195 on our setup because the CPU keeps evicting the sets starting from the 196th. We chose to call this cache area the "CPU deadzone" as the core's activity prohibits us from placing any information there as it gets evicted a few cycles after being written. Depending on the code's size, the CPU "deadzone" might change: the longer the code, the bigger the zone.

This effect had to be taken into consideration in order to successfully carry out the initial attack. According to our observations, and the way we implemented our Trojan code, the evictions caused by the Trojan's activity in the data cache are successive at the level of the cache sets. This means that there is a recognizable pattern we are able to exploit to separate the "useful cache misses" from the ones related to the "CPU deadzone". Once our spy code recovered all of the cache misses that occurred, we only keep the sets where evictions happened that are adjacent to each other.

The fact that we were able to extract data from a given process to another process, transiting through the cache, shows that the CVA6 core is vulnerable to micro architectural threats. These observations motivated the second type of experimentations presented in section IV.

For instance, we wondered to what extent we could predict how noise, either because of the CPU's activity or another untargeted process, would affect the previous attack, and how to modify it accordingly. In addition, we wondered if scheduling could also generate noise affecting the attack's behavior.

D. *About the challenges to tailor a micro architectural attack to a specific hardware/software scenario*

Every attack scenario (combination of a targeted hardware and a software victim code) is different and implies a different CPU behavior. More specifically, the way the micro architectural elements react to the code's execution will also differ based on the hardware/software combination.

Therefore, in order to carry out an attack at the micro architectural level, there are several key points to take into consideration in order to maximize its success rate. First, the knowledge about the targeted victim application is crucial. Most micro architectural attacks need to be "triggered" at a precise timing (when a specific instruction is issued by the victim code, computing on secret data for example). Moreover, this also implies that it is required to know the nature and the structure of the secret one wants to extract. This information will drive the choice for a pertinent encoding technique in order to maximize the information extraction.

Then, a precise knowledge of the targeted micro architecture is also necessary. It does not need to be exact or

at a very-low granularity level. Sometimes, only a general intuition of how it works is enough, as it was the case in the Branchscope attack [14]. This knowledge is generally gained thanks to reverse-engineering effort, because most micro architectural elements are "black box" and undocumented. However, in the case of an open-sourced CPU such as CVA6, this knowledge is easier and faster to get, which can be considered as a leverage. We did put this into practice with our presented attack as we reverse engineered the CVA6's data cache using both the available source code (which sped up the process) and our observations. Knowing how the exploited cache is being filled and emptied is part of the required knowledge to build a successful micro architectural attack. As such, for an attacker targeting an unknown cache implementation without any implementation details, the preliminary reverse engineer study would require much more efforts. An open-source implementation helps the attacker, but even in such favorable conditions, the reverse engineering effort proved to be quite time consuming.

IV. Taking into account scheduling and noise

A. Augmented threat model

Let us now introduce a new threat model, slightly modified compared to the previous one (section II.A.). The goal of this new threat model is to present a use-case situation, closer to a real-life scenario.

In the following, the term "noise" refers to an unwanted (for the attacker) software activity affecting the cache and thus potentially reducing the attack's effectiveness. Scheduling is one possibility to introduce such noise and supposedly reproduces an OS' behavior. The main objective is to characterize the presented micro architectural attack in a noisy and "scheduled" environment in order to anticipate how it would behave when run on an Operating System (OS), and how to take these observations into consideration for an attacker. In this second threat model, we now have a running skeleton of a minimalist OS that encapsulates both the victim and the attacker processes and places them in two separate domains: a confined and an unconfined one. By design, the two domains cannot share information directly, but they do share the same micro architectural structures.

B. Impact of a scheduler

The first modification consisted in adding a scheduling process to the experimental setup, as shown on Fig. 5. The idea here is to mimic the scheduling implied by the use of a potential OS in the threat model. The purely sequential process execution shown in Fig. 2. has therefore been replaced by a very simple simulation of scheduling.

```
Prime+Probe_simple_scheduling(void)
1   Run 60% of priming_phase()
2   Run 50% of victim_process(secret_value)
3   Run the 40% remaining of priming_phase()
4   Run the 50% remaining of victim_process(secret_value)
5   If noise_generation = 1
6   Then run 100% of  eviction_perturbation(quantity_of_noise)
7   Run  100% of the probing_phase()
8   Return(secret_value)
```

Fig. 5. Pseudocode of an example of the simply scheduled attack. The % values are arbitrary and can be modified, just as the order of execution, in order to be able to test any and every possible scenario.

With our scheduler, we manually choose the allocated execution time for each function on a mono threaded mono core processor. Here, only the victim and the attacker processes are running. We will discuss the introduction of new processes in the next section.

Scheduling introduces new issues to carry out the Prime+Probe attack. We needed the attack to be triggered at the appropriate timing. The ideal timing is just after the Trojan's execution. If the attack is executed before the Trojan's encoding, the attacker is unable to extract anything, as the secret information has not been brought to the cache yet. If it is executed too late, data in the cache is likely to be overwritten quickly.

Moreover, the computation time of the victim and the scheduling process needed to be known and the attack had to be adapted to it. More specifically, the execution of the priming, the victim and the probing had to be adapted not to exceed their given timeframe. Otherwise, the scheduler switches context, and the CPU runs another process thus evicting the secret data to extract.

The main idea to account for this issue is to implement strategically our attack code in order to make it as fast as possible. Encoding by filling completely or not a single cache set (instead of 256) is an appropriate solution. It reduces the possible values to extract from 196 to only two (0 if the set is empty or 1 if it is filled) but makes each iteration much faster. It implies several iterations to extract a secret value bigger than 1 bit. This encoding can be used practically in real use-cases to extract a single bit of the nonce in an ECDSA cryptographic algorithm which is enough to perform an attack [15]. One of the easiest methods to speed up the attack's computation is thus to choose the most efficient cache-encoding technique, with respect to the capability of extracting the whole secret.

C. Noise generation and effects on the attack

One last important effect of introducing an OS to our threat model is the addition of perturbations. Indeed, in the context of an OS, several different processes are being constantly computed concurrently. This will generate what can be considered as "noise" to the attack scenario. This noise will in fact cause unknown and unwanted evictions inside the cache that might hinder the attack's capacities. We studied the impact of the presence of an additional process generating noise to our implementation. We propose here an analysis of the effects of this noise on the attack code and recommendations to avoid being impacted by it.

When adding our additional noise-generating process, we chose to consider the worst-case scenario: a process that would cause as many random cache evictions as possible during its allocated timeframe. To this end, we implemented a pseudo-random generator using an AES [16] algorithm. We used those pseudo-random numbers as addresses to evict from the cache. The contents at these addresses are replaced with data unrelated to the attack. The additional process is run between the Trojan's execution and the spy's probing phase, otherwise it would not affect the attack.

When considering the first encoding technique used in this work (extracting values from 0 to 255 based on the cache sets), such a random-evicting process has very significant effects. Fig. 6 gives the evolution of the success rate of the attack in

function of the number of evictions caused by the additional process.

The second encoding considered earlier (with only 2 values possibly extracted, 0 or 1) is much more robust facing noise as it only uses a limited amount of space inside the data cache compared to the first one. These results show that the attack's resistance to noise is depending on the encoding technique used. Moreover, to make a micro architectural cache-based covert-channel more resilient towards noise, it is recommended to study, when possible, the CPU's and the victim's behavior in terms of cache usage in order to target specific areas that are the least used ones, so that it maximizes the attack's success rate.

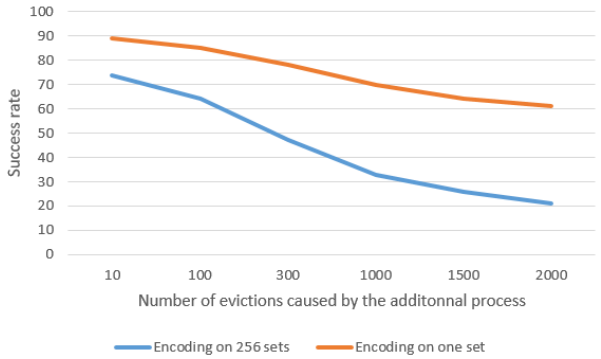

Fig. 6. Evolution of the success rate (in %) of the 2 considered encoding solutions compared to the number of evictions caused by the additionnal process.

D. Combination of scheduling and noise generation

Our last experimentation consisted in combining a process that is causing random cache evictions based on a process performing an AES encryption similarly to the experimentation presented earlier, and the scheduling process presented in section IV. B. The results obtained give a clear vision of the overall quantity of perturbations an attacker has to take into consideration when trying to run a cache-related attack in a noisy environment, without any privileges to reduce it. Fig. 7. shows the obtained indexes of the sets where cache misses occurred.

Out of the 256 cache sets available on the CVA6 standard data cache configuration, 111 sets were modified between the priming phase and the probing phase of the attack. The causes and repartitions of these sets are detailed further on Fig. 7. The useful information placed by the Trojan here is "30" as we have a total of 30 cache sets that were modified by the Trojan in a chosen range. This leaves 81 cache sets that were modified during the attack by processes other than the victim or the Trojan. The superposition of scheduling and random noise generation causes the implementation of the attack to be even more challenging.

It is therefore crucial to be able to differentiate the evictions that were caused willingly by the attacker from the rest that is related to the activity of other processes. To achieve this, it is required to apply some of the recommendations developed in section III.D. Some reverse engineering efforts enabled us to identify and characterize the "CPU deadzone" in this precise use-case. This zone is always in the same range of cache sets, and is only modified when one of the codes is modified, making its identification easier than for the other eviction sources.

The most difficult step consists in sorting out the useful information from the noise generated by the presence of the process causing random evictions. Its activity can be divided into 2 categories: the effective evictions caused by the code we implemented, these are pseudo-random; and the evictions caused by the presence of an encryption table when it is accessed. We will now detail how we proceeded in our specific scenario, considering we know exactly what happens during our attack.

The former can be identified, provided it does not "blend in" with the useful information. Indeed, the sets that are impacted by these pseudo random processes stand out when considering all of the sets that were impacted during the time of the attack, as they are isolated from the rest. There is still a possibility for these evictions to occur in a set that is successive to the "useful sets" that were modified by the Trojan. In that case, the attack fails as we recover a value that is higher or lower by one to the expected value.

The latter can be characterized through some reverse engineering. To this end, we carried out several attacks without running the victim code, meaning that the Trojan does not cause any evictions inside the cache. We were then in capacity to determine the cache sets that were modified by the operations on the encryption tables. These sets are the same from an experiment to another, and only change when the tables are modified.

By combining all of the previous observations, we were able to recognize the pattern caused by the Trojan amidst the rest. It consists in a successive pattern, that is located at a range outside of the "CPU deadzone" and the encryption table's activity that are both definite and not changing from an experiment to another. There is still a small chance for the process generating random evictions to make the attack fail, and we are not able to detect it. This chance is relatively small as the noise-generating process is limited and can only proceed to a few evictions in the timeframe given by the scheduler. Generally, this process generates between 5 and 10 evictions during a single timeframe according to our implementation of the scheduler and noise-generating process.

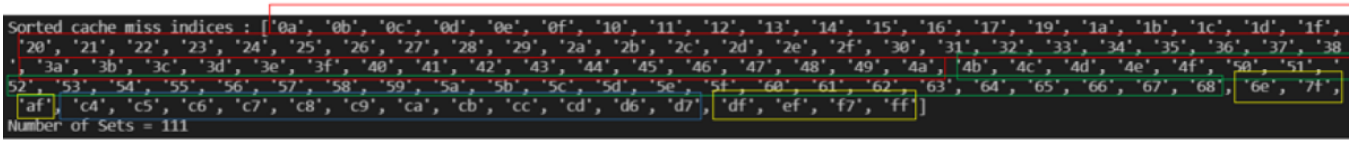

Fig. 7. Overview of the different cache sets indexes where cache misses have been obtained in a "semi-realistic" use-case. The sets circled in red have been filled with data related to the AES' encryption table. The sets circled in green are filled with useful information for the attack, placed by the Trojan. The sets circled in yellow are the sets filled by our random noise generators. The sets circled in blue are the sets in the CPU deadzone containing CPU opeartions and various other intermediate operations. A total of 111 sets were modified after the priming phase in this example.

When attacking in a more realistic scenario, the attacker does not know about the structure of the other processes that are running and causing perturbations. Therefore, sorting out the different sources of evictions can prove to be challenging, depending on the other processes' behaviors.

Similarly to the method that we applied during our experimentations, reverse engineering efforts are required. Proceeding to several "calibration measurements" will yield interesting outputs about the behavior of the processes running concurrently with the victim and the victim itself. This information can then be used as an input for the attack. This only applies to an unprivileged attacker, as a privileged one can prohibit unwanted evictions caused by other processes while the victim is being run. This can either be done by moving other processes to another core (when available), or by modifying the priority levels of the different processes.

V. Conclusion and Future Work

We presented a proof of concept implementation of an access-driven cache-based micro architectural covert channel on the RISC-V CVA6 core, in a baremetal context. Micro architectural attacks have to be mitigated at the micro architectural level, otherwise the risk will still exist.

Moreover, we also detailed how to adapt a given attack to a specific threat model and how to take into consideration the effects of scheduling and noise to maximize the attack's effectiveness. This analysis can be used in the context of a realistic model to carry out a covert-channel with a running OS to produce a more dangerous attack with higher chances of successful extraction.

With more knowledge about the target and the victim comes more power for secret extraction. Open-source implementations of both hardware and software modules come at the price of new challenges for the designer to propose a secure platform.

A future work is to carry out the same attack on a rich OS (such as Linux) and to compare the outputs of both. We will then be able to list which conditions favor the effectiveness of a micro architectural attack in a more complex scenario so that we would be able to propose counter-measures to mitigate the micro-architectural attacks in this context.

References

[1] "Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain," JSOF, [Online]. Available: https://www.jsof-tech.com/ripple20/. [Accessed 17 september 2020].

[2] J. V. Bulck et al., "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, 2018.

[3] C. Canella et al., "Fallout: Reading Kernel Writes From User Space," *26th ACM Conf. on Computer and Commubications Security*, 2019.

[4] S. van Schaik et al., "RIDL: Rogue In-Flight Data Load," *40th IEEE Symposium on Security and Privacy*, May 2019.

[5] M. Schwarz et al., "ZombieLoad: Cross-Privilege-Boundary Data Sampling," *CCS 19: Conf. on Computer and Communications Security*, pp. 753-768, 2019.

[6] Z. Allaf et al., "A Comparison Study on Flush+Reload and Prime+Probe Attacks on AES Using Machine Learning Approaches," *UK Workshop on Computational Intelligence,* pp. 203-213, 2017.

[7] Y. Yuval and F. Katrina, "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 719-732, 2014.

[8] D. Gruss et al. , "Flush+Flush: A Fast and Stealthy Cache Attack," *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment,* vol. 9721, pp. 279-299, 2016.

[9] C. Disselkoen, D. Kohlbrenner, L. Porter, D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," *26th USENIX Security Symposium (USENIX Security 17)*, pp. 51-67, 2017.

[10] "CVA6 Core," ETH Zurich, [Online]. Available: https://github.com/openhwgroup/cva6. [Accessed 27 06 2022].

[11] V. Martinoli, Y. Teglia, A. Bouagoun and R. Leveugle "CVA6's Data cache: Structure and Behavior", *arXiv e-prints: 2202.03749*, 2022, [Online]. Available at: https://arxiv.org/abs/2202.03749.

[12] "Verilator," Veripool, [Online]. Available: https://www.veripool.org/verilator/. [Accessed 27 06 2022].

[13] "Spike RIDC-V ISA Simulator," RISC-V, [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim. [Accessed 27 06 2022].

[14] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," *ASPLOS'18*, Williamsburg, VA, USA, 2018

[15] D. F. Aranha et al., "*LadderLeak*: Breaking ECDSA With Less Then One Bit Of Nonce Leakage," *CCS 20: Conf. on Computer and Communication Security*, pp. 225-242, 2020.

[16] Dworkin, M. , Barker, E. , Nechvatal, J. , Foti, J. , Bassham, L. , Roback, E. and Dray, J., "Advanced Encryption Standard (AES)", *Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology*, 2001, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.FIPS.197 (Accessed May 4, 2022)