Journal of
Low Power Electronics and Applications

MDPI

Article

CCALK: (When) CVA6 Cache Associativity Leaks the Key

Valentin Martinoli [1,2,*], Elouan Tourneur [1], Yannick Teglia [1] and Régis Leveugle [2]

[1] Thales DIS, 13600 La Ciotat, France
[2] University Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering University Grenoble Alpes), TIMA, 38000 Grenoble, France
* Correspondence: valentin.martinoli@external.thalesgroup.com

Abstract: In this work, we study an end-to-end implementation of a Prime + Probe covert channel on the CVA6 RISC-V processor implemented on a FPGA target and running a Linux OS. We develop the building blocks of the covert channel and provide a detailed view of its behavior and effectiveness. We propose a realistic scenario for extracting information of an AES-128 encryption algorithm implementation. Throughout this work, we discuss the challenges brought by the presence of a running OS while carrying out a micro architectural covert channel. This includes the effects of having other running processes, unwanted cache evictions and the OS' timing behavior. We also propose an analysis of the relationship between the data cache's characteristics and the developed covert channel's capacity to extract information. According to the results of our experimentations, we present guidelines on how to build and configure a micro architectural covert channel resilient cache in a mono-core mono-thread scenario.

Keywords: hardware security; micro architecture; covert channel; cache; timing side-channels; RISC-V; CVA6; Linux

Citation: Martinoli, V.; Tourneur, E.; Teglia, Y.; Leveugle, R. CCALK: (When) CVA6 Cache Associativity Leaks the Key. *J. Low Power Electron. Appl.* **2023**, *13*, 1. https://doi.org/10.3390/jlpea13010001

Academic Editors: Teresa Cervero, Kevin Martin, Mario Kovač and Maurizio Martina

Received: 27 October 2022
Revised: 19 December 2022
Accepted: 23 December 2022
Published: 27 December 2022

1. Introduction

Recent CPUs embed optimization mechanisms that are exploited by micro architectural attacks to cause information leakages (e.g., [1] taking advantage of out-of-order execution). The root cause of micro architectural covert channels is the competitive access to shared and limited hardware resources. According to these principles, we implemented a micro architectural covert channel capable of extracting information through the L1 data cache of the CVA6, a 64-bit open-source application class RISC-V processor. We applied this covert channel in a realistic scenario targeting a software implementation of the AES algorithm on a FPGA instantiation of the CVA6 running a Linux OS. We discussed the experimental results we obtained and their limitations. We showed the different challenges of adapting such a covert channel to a given hardware-software scenario. Moreover, we showed the implications of the operating system on the attack. We concluded that micro architectural covert channels are practical and therefore a real threat on a RISC-V platform in a realistic context, especially when using open-source materials.

We made the following contributions:

1. Proposing an implementation of an access-driven, cache-based, micro architectural covert channel on the CVA6 platform embedding a Linux OS;
2. Showing the root cause of the covert channel on the considered platform as well as its limitations;
3. Applying the covert channel to a realistic use-case on an AES implementation to show it is practical and could be exploited in a real-life scenario;
4. Showing the impact that the hardware-software environment has on the covert channel and their relationship with the level of vulnerability of a given cache structure.

Section 2 details the type of covert channel used and the processor architecture. Section 3 summarizes the implementation of the attack and analyses results. Section 4 shows results when the attack is applied to AES, before the conclusions.

2. Background

2.1. Micro Architectural Cache Covert Channels

Covert channels are specific methods used to create a capability to transfer information between entities (e.g., processes) that are not allowed to communicate according to the security policy. These unwanted communication channels are hidden from the access control mechanisms of operating systems, even when intended to be secure. The addition in the most recent high-end CPUs of many optimization mechanisms has led to the complexification of the underlying micro architecture. Consequently, several covert channel possibilities have been uncovered relying on micro architectural elements (e.g., cache memories, internal buffers...).

Traditionally, covert channels are known to enable an attacker to transfer information between processes that are isolated from each other, according to the running security policy. However, the recent development of the micro architectural attacks such as Foreshadow [1], the more recent MDS (Micro Architectural Data Sampling) attacks [2–4] and Meltdown [5] conducted to a new type of covert channel. All of these attacks are concluded by micro architectural covert channels that can transfer information from microarchitectural structures, to the architectural world where it can be observed. Thus, studying covert channels in an effort to mitigate them is valuable, as protecting a system from this threat would result in the inability for an attacker to extract secret information. Indeed, in a resilient system against covert channels, the attacker could still gather the information at the micro architectural level. However, they would then be unable to observe it, as the gathered data would remain in the micro architectural structures, without any possibility to retrieve it. The access-driven cache-based mechanism proved to be the most commonly used type of micro architectural covert channel in the recent attacks. The caches are an ideal target because they are shared among different processes. In theory, all caches (instruction, data) of all levels (1, 2, and even 3 when it exists) are subject to these threats; however, the L1 data cache has been the favored one in the literature as it is also the easiest to exploit since it is closer to the CPU. Moreover, it is also the most easily accessible micro architectural structure. There are many cache covert channel variants in the literature such as Prime + Probe [6], Flush + Reload [7], Flush + Flush [8], Prime + Abort [9].

The sole purpose of a cache is to provide increased performances. However, it can be exploited as a side channel in a malicious way to cross security boundaries between processes. All of the variants cited above are relying on the ability for an attacker to infer whether a specific cache line has been evicted or not. Using the principle of micro architectural covert channels, this capacity can then be exploited to deduce information about the secret to be extracted. The access-driven cache-based covert channels are often compared to time-driven covert channels as the leakage also originates from timings considerations. However, access-driven cache-based covert channels are focusing on the cache's behavior and leverages it more precisely. Time-driven covert channels focus on evaluating the overall execution time, whereas access-driven covert channels rely on the ability to detect whether a cache line has been evicted or not using timing measurements.

Previous works [10,11] by Wistoff et al. considered the micro architectural cache covert channel threat on CVA6. These works focus on the development of the fence.t instruction as a mitigation to these attacks. This instruction introduces the possibility to clear the leaky micro architectural elements and cause a context switching. These works proposed an analysis of a toolkit Prime + Probe covert channel on an FPGA emulated CVA6 core running a seL4 [12] microkernel. While Wistoff et al. focus on developing a mitigation to toolkit cache covert channel attacks, this paper proposes an implementation from scratch and details the challenges and methodology of implementing such a covert channel on a

specific core. It also studies the beneficial and unbeneficial factors and conditions for an attacker to carry out such an attack successfully.

2.2. The CVA6 Core and Its Data Cache Structure

CVA6 is an open-source core that has been developed by ETH Zürich and University of Bologna [13]. It is an application-class 64-bit processor, implementing the RISC-V Instruction Set Architecture [14]. The RTL (Register-Transfer Level) description is written in System Verilog. This core was chosen as a target for our experimentations because it is Linux-capable, and can run the M, S, and U privilege modes. As part of its hardware package, the CVA6 embeds a Translation Lookaside Buffer (TLB) and tightly integrated data and instruction caches. The core was optimized for performance, reaching frequencies up to 1.7 GHz, but security was not the focus of the design. Its pipeline was made of six stages as shown in Figure 1. The CVA6 was almost totally in-order; however, the write-back happened out-of-order inside the execution stage. More details about the CVA6 core are available online [15].

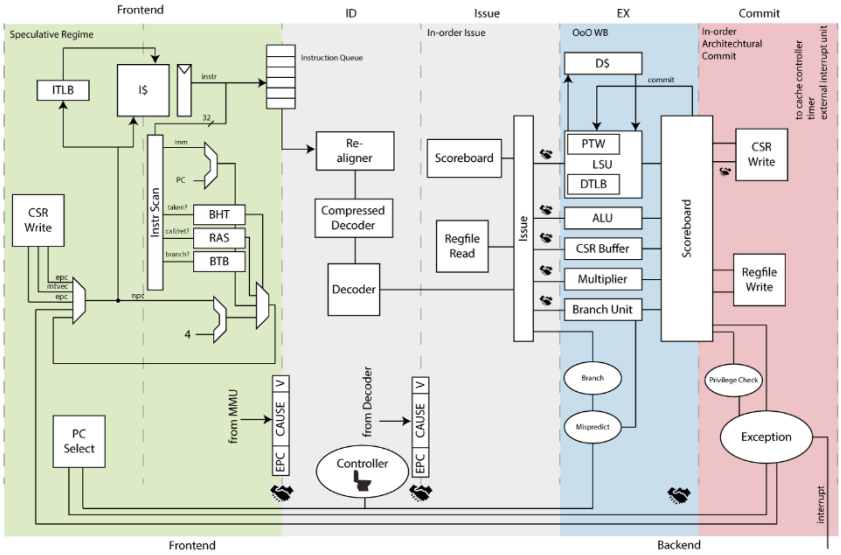

Figure 1. Representation of the CVA6 6-stage pipeline [15].

By default, the data cache is a 32 KB 8-way set-associative cache. It contains 256 sets, each composed of 8 ways of 16 bytes. The terms ways and cache lines are equivalent. For the remainder of the document, the term "way" is therefore used. Its default filling policy is write-through no write-allocate.

Write-through means that the cache controller updates the cache and the main memory synchronously upon every write access request to a memory block. When a piece of data does not reside in the main memory, the no-write-allocate policy consists of writing the data only in the main memory and not in the cache. It implies that a piece of data is loaded into the cache exclusively on read misses. With these policies, a write operation of a given data not already allocated in the data cache causes this piece of data to be written in the main memory only. When this piece of data is accessed for a read operation, as it has been allocated in the main memory, but not in the cache, it will cause a cache miss. This will result in the data finally being brought inside the data cache, according to the no-write-allocate policy. Moreover, the write-through policy also implies that upon writing the data inside the data cache, the cache controller also updates the value of the data inside the main memory so that it is always coherent with the data cache.

It is worth noting that cache addresses in the CVA6 core are physical, and not virtual. Consequently, the number of bits required to address the data cache varies with the cache's size, as there is a need to address each possible cache block individually. When referring to a cache hit, we consider a request to a data that has already been allocated in a memory

block inside the data cache. A cache miss is considered to be a request to a piece of data that has no allocated memory block inside the data cache.

Regarding the eviction policy, the CVA6 uses an LFSR (Linear-Feedback Shift Register) to select the specific cache line index to evict. This 8-bit parametric LFSR allows selecting which cache line has to be evicted based on a pseudo-random basis. When a cache miss occurs, the LFSR is used if all the cache lines are unavailable for a given set (i.e., already containing data). Otherwise, if there are some invalid cache lines (e.g., containing outdated data, or no data at all), the line with the lowest index having its validity bit equal to zero is selected. More details about the CVA6's data cache implementation are available in the literature [16].

3. Building a Covert Channel on the FPGA-Instantiated CVA6 Running Linux

3.1. Initial Threat Model

The applicative scenario we chose is detailed here. The victim application we targeted ran computations implying a secret value. We considered that it is isolated from other processes for security purposes preventing any eavesdropping by software means. We considered this victim application to be compromised either by a library containing a Trojan or a buggy implementation causing leakages at the micro architectural level. This strong hypothesis is legitimate as the recent Ripple20 series of vulnerabilities [17] has shown it: a series of critical vulnerability was discovered in a widely used TCP/IP library deployed in a vast range of applications. We consider an attacker that is aware of this leakage and willing to recover the secret used by the victim process. The secret information is recovered using the micro architectural leakage, crossing the security boundaries introduced by the logical isolation.

The technical target consists in a mono-core and mono thread scenario. The mono-core scenario first adds some difficulty for a potential attacker as they cannot take advantage of high-end optimization mechanisms (out-of-order execution, Simultaneous Multi-Threading...) to gather information. These mechanisms are widely used in micro architectural attacks to recover data from a neighboring process located on another logical core. More generally, a simpler core design implies a restricted number of attack paths available. It also reduces the available covert channel techniques that are applicable, as several of them require the use of the mechanisms named above. However, even if a multi-core scenario causes a greater attack surface, the exploited hardware resources are shared among several cores. This implies a significantly higher amount of perturbation for the attacker trying to leak data from these resources. In the specific case of caches, the previous logic still applies with a variation induced by the cache's size. Even if there is more contention in a multi-core system, if the cache's size is sufficient, there might be no additional contention compared to a mono-core system.

In a mono threaded use-case, the execution time is shared only by a single processing thread. It results in more time for the attacker to leverage a covert channel and less chances of being interrupted during the attack.

The victim runs on a CVA6 that runs an operating system (OS) on top of which runs the victim within a first domain, considered to be trusted. This victim program is assumed to be compromised by a Trojan trying actively to leak the secret information from the victim application.

The attacker itself is contained in a second untrusted security domain. They time-share the core with the victim application and run a spy program that tries to recover the information leaked by the Trojan. The threat model is summarized in Figure 2. Both the victim and the attacker applications run in the user land. Indeed, CVA6 does enable the use of timestamp instructions in user mode. It might not be the case for other CPUs where developing kernel modules would therefore be required to access those necessary instructions in order to carry out a time or access-driven cache covert channel.

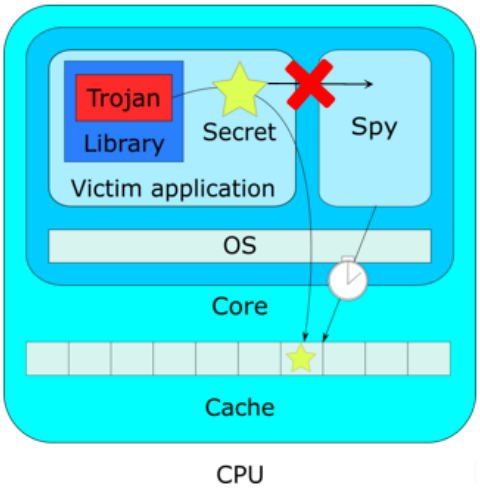

Figure 2. Chosen threat model for the micro architectural covert channel with a running OS.

3.2. Experimental Setup

We now detail the setup we used during our experimentations. For the hardware configuration, we chose to work with the default version of CVA6, as directly available from the Open Hardware Group's Github repository [15]. More specifically, this meant that the core we worked on used 64-bit addresses, and had the default cache configuration corresponding to the one introduced in Section 2.2.

We instantiated this CVA6 on a Genesys 2 FPGA platform from Digilent [18], as there were a lot of tools and configurations available to work with. The Digilent Genesys 2 board was based on the latest Kintex-7™ Field Programmable Gate Array (FPGA) from Xilinx and is a high-speed FPGA. We communicated with the FPGA platform using a Linux computer through SSH and using the Ethernet port available on the Genesys 2 board.

For the software configuration, we relied on the toolchain contained in the Open Hardware Group's Github repository to compile and synthetize the RTL into an exploitable bitstream and memory configuration file. We then used the Xilinx's Vivado Design Suite [19] to instantiate the results of the synthesis on the FPGA board. We then selected a Linux OS, generating our own Linux image using the toolchain available on CVA6-SDK Github repository [20]. It was based on a pre-built image directly available on the same repository, but we slightly modified it to fit our specific needs, adding support for SSH communication. The Linux image is lightweight, and we chose not to add any tool or process that could hinder our experimentations. The Linux image is then embedded in a standard 64 Gb SD Card and inserted inside the Genesys 2. The OS will then be loaded at each reset of the FPGA board.

3.3. First Experimentations on Information Transmission and Statistical Analysis

To carry out a micro-architectural covert channel, it is beneficial to have a good picture of the targeted cache architecture and its mechanisms. A first advantage for an attacker targeting an open-source core is the availability of the sources of the targeted data cache. The reverse engineering efforts are considerably sped up in an open-source scenario, even if this effort remains not negligible. In our case, we leveraged the work done in [16] where the authors detailed both the cache structure and its implementation.

We further carried out several preliminary experiments in order to identify a potential encoding technique e.g., a specific method to place the secret information inside the data cache in order to transmit it. Our initial idea was to propose an implementation of the Prime + Probe covert channel, therefore we began by observing the cache misses. A simple application causing cache eviction by filling an array structure was used as a preliminary victim. A second application measured the time it takes to access its own data before (when it was in the cache) and after the victim has been run (when the data has supposedly been evicted) similarly to the prime and probe steps of the related covert channel. These tests

aimed at two different objectives: confirm our comprehension of the cache's implementation and achieve a first level of information transmission via controlled cache evictions.

Based on the same idea as the experiment described above, several tests and variations of this experimentation have been made in order to establish a correlation between the activity we caused with a given code (the victim filling an array with data), and the resulting evictions. These experiments included filling different sizes and types of array structures (smaller than the cache, same size as the cache, bigger than the cache for example) for causing different cache eviction patterns. We then measured where the evictions happened inside the cache using the second application. The understanding of the hardware implementation of the data cache was a valuable input for these first experiments. We started out by running a skeleton of Prime + Probe covert channel consisting of an attacker process (or spy process) filling the cache with its own data (Prime step) and a second process (or victim process, containing the Trojan) trying to cause evictions equal to a given secret value that was hardcoded inside it. We then proceeded to timing measurements inside the spy process (that carried out the Prime step) using the RDCYCLE RISC-V instruction [21] in order to observe the different cache evictions caused by the victim process and its Trojan.

The spy process also generated a log file containing all of its measurements that we statistically analyzed thoroughly after using Python scripts. This statistical analysis over a large number of experimentations is a mandatory step compared to a BareMetal context as the OS causes multiple unwanted cache evictions because of the concurrent processes that are running. Considering multiple experimentations and subtracting the results of the experimentation without the victim process running (normal activity only with the OS running) enabled us to reduce the impact of this unwanted "noise" in our measurements, and observe only the useful evictions caused by the Trojan contained in the victim process.

After these experiments, we were able to observe a difference between the cases in which the victim process creates evictions and the other cases. Recognizable patterns caused by the victim process' activity (generating cache evictions) showed that it was possible to transmit information, even if the OS adds some "noise" that could hinder the transmission. In this specific case, the information transmitted consists in the binary decomposition of the secret value, encoded in the amount of cache evictions caused by the victim application containing the Trojan. The secret value's binary decomposition is transformed into a specific cache contention pattern (further details about the contention pattern are given in Section 3.4). The Trojan then causes the corresponding cache evictions to match this pattern that is then recovered by the attacker during the prime phase of the covert channel. More cache evictions than a fixed threshold (determined during preliminary experimentations to differentiate the cache hits from cache misses) imply a value of 1, less evictions than the threshold means the recovery of a 0.

It also enabled us to create a set of analysis tools to minimize the impact of the OS on the observed traces and visualize the results of our minimal Prime + Probe covert channel. The next step consisted in establishing a covert channel, enabling us to transmit a given value through the data cache.

3.4. Implementation of a Basic Tailored Covert Channel on the CVA6

To build a micro architectural cache covert channel, it is mandatory to be able to recover a precise value hidden by the Trojan inside the data cache. According to the previously presented observations, and the understanding of the data cache's structure, we built a covert channel on the CVA6 capable of transmitting up to 256 bits of information with the default data cache configuration.

Applying the same code structure as in Section 3.3 inspired by the Prime + Probe technique, Figure 3 gives the pseudocode of the improved version of our initial covert channel.

```
Prime(table[])
    FOR i in range(0, size(table)-1)
        filling_table[i] = 400
        temporary_variable = table[i]
    ENDFOR

Probe(table[])
    FOR i in range (0, size(table)-1)
        t1 = RDCYCLE
        temporary_variable = table[i]
        t2 = RDCYCLE
        IF (t2-t1 > SEUIL)
            Miss_count[i] = Miss_count[i] + 1
        ENDIF
    ENDFOR

Trojan(secret)
    Trojan_table[CACHE_SIZE]
    FOR i in range(0, size(Trojan_table)
        IF (i % NbSets == 0)
            FOR j in range(0,Size_binary_secret)
                IF (Binary_secret[j] == 1)
                    Trojan_table[i+j] = 400
                    temporary_variable = Trojan_table[i+j]
                ENDIF
            ENDFOR
        ENDIF
    ENDFOR

Victim(secret)
    ...
    trojan(secret)
    ...

Spy()
    For i in range(0, NB_ITERATION)
        For j in range(0, NB_SAMPLES)
            Prime(spy_table);
            Victim();
            Probe(spy_table);
        ENDFOR
        Generate_logs();
    ENDFOR
```

Figure 3. Pseudocode for our Prime + Probe micro architectural covert channel; the spy function is the main one and calls the victim function containing the Trojan.

Considering the characteristics of the data cache, the spy process builds an extraction array fitting the cache's structure. This array contained 2048 cells of 16 bytes, analogously to the CVA6's data cache. This array was used to fill the cache during the Prime step, made by the spy process. As the cache used a no-write-allocate policy, it is necessary to cause a read-miss on every element of the extraction array in order to bring them inside the data cache. To this end, we sequentially allocated and then read back each cell of the array.

Once the cache was filled, the spy process created a thread running the victim and the Trojan it contains. We discuss the realism and the applicability in a real-life context of this scenario in a later section. For this iteration of the experimentations, the victim simply consisted in an addition and the secret was the result of the operation, or even just a hard coded value acting as a secret to be extracted. The rest of the victim code is composed of the Trojan that will cause targeted evictions inside the data cache. In order for the evictions to be effective and transmit the desired value, it is required to take into consideration the structure and behaviour of the data cache.

We noticed that working with the cache sets produced the expected results, and was easier to implement than working at the cache way granularity, mainly because of the pseudo-random eviction policy. The secret value was transmitted bit by bit, meaning that a cache miss was interpreted as a value of 1, whereas a cache hit was interpreted as a value of 0 (because the spy's value was not evicted).

The Trojan proceeded similarly to the spy process during the Prime step and was allocated then read again cells in another array (called "Trojan's extraction array") with

the exact same dimensions as the cache. However, the Trojan did not fill every cell of the array, but only the cells with indexes that corresponded to a value to be extracted equal to 1. It ignored cells with indexes corresponding to a value to be extracted equal to 0. The main idea was that filling specific regions (bit of the secret equal to 1) of the Trojan's extraction array directly translated in a cache miss for the spy inside the corresponding cache sets inside the data cache while for the untouched regions (bit of the secret equal to 0) it translated in a cache hit. Figure 4 represents the situation and shows the corresponding evictions caused by the Trojan's action.

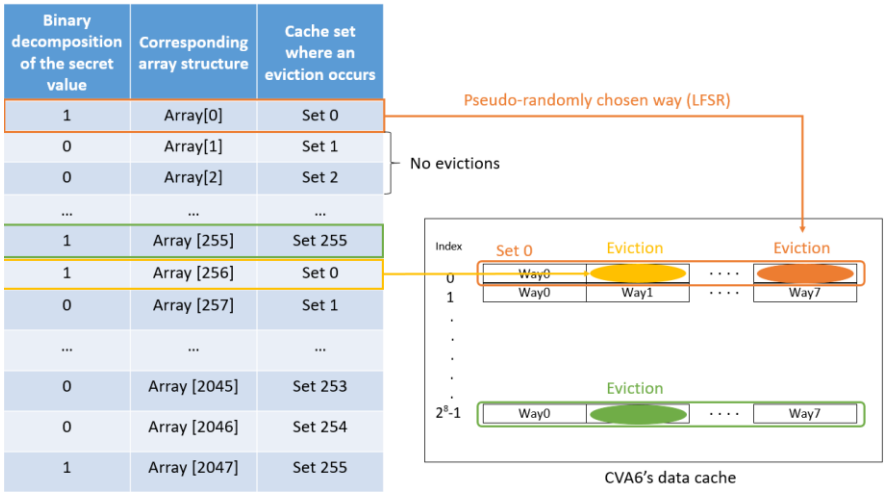

Figure 4. Detail of the correspondences between the value to encode, the extraction array's structures, the cache sets where evictions occur, and the state of the CVA6's data cache.

In practice, the Trojan therefore fills its extraction array at specific locations to cause evictions in the corresponding cache set. Given that we wanted to extract a value equal to 1 as the first bit in the secret's value decomposition, the Trojan therefore filled its array cells having indexes that verify:

$$Index \equiv Targeted\_set \ mod(Total\_number\_of\_sets)$$

Hence, in the default cache configuration we studied throughout this article, and for the first bit of the binary decomposition (thus we target the set number 0) this translates to:

$$Index \equiv 0 \ mod(256)$$

The value written inside the cells does not matter at all, as only the cache eviction is useful for the covert channel. Each time the Trojan fills one of the cells, it causes a corresponding eviction inside the set having an index equal to the second term of the previous equation. For our example, the eviction was caused in the set 0. There were 8 possible ways to be replaced inside each set, and we could not easily predict, or choose which one was evicted. This is the reason for choosing a "cache set encoding" technique instead of working at cache line granularity that proved to be harder since the choice is made pseudo-randomly by the LFSR. We had no guarantee however that the evictions caused inside a given set will occur on different cache ways. It might happen that the way 0 inside the set 0 is evicted twice for example (as an example, when filling Array [0* Total_number_of_sets] and Array [2* Total_number_of_sets]). Overall, and on several repetitions of the covert channel, we are capable of distinguishing data sets where "a lot" (will be quantified in the next section) of evictions occurred, caused by the Trojan, compared to data sets where no evictions were caused and the spy's data are still intact, leading to the reconstruction of the secret value.

This reconstruction was made possible by the spy process' measurements during the Probe step. Once the Trojan filled its extraction array according to the secret's binary decomposition, we ran the spy process again. It measured the time it takes to access again to all of its data. We were then capable to distinguish data that have been evicted from data that have not been evicted, using the RDCYCLE instruction. Indeed, some cells inside the spy's array experienced a higher access time compared to the others; thus, they were evicted from the cache. Using these measurements and our Python script that processed the logs generated by the spy process, we reconstructed the secret value to be extracted. Experimental results proving that our attack implementation was practical on a simple applicative example are given in the next section.

To summarize, the main aspect of this covert channel consisted in translating the binary decomposition of the secret value to a corresponding contention inside the matching data sets. Having a value of 1 to extract at a given index, called p, in the binary decomposition, meant that we wanted to create contention inside the cache set number p. For that purpose, the Trojan fills all of the cells in *Trojan_table* that will cause evictions inside the set number p. By experimenting, we found out that these cells are the ones verifying: $Index \equiv p \ mod(Total\_number\_of\_sets)$. Filling these cells with Trojan's data caused an eviction of the Spy's data inside the set number p. These data were previously allocated inside the cache during the Prime phase. Therefore, when the Spy Probes its data again to verify the time it takes to access it, it can conclude about the sets that contain contention or not by looking at the amount of cache misses for every cache set. According to the evictions caused by the Trojan's activity, a high number of cache misses for the set number p is identified. This means that the Spy process will experience a higher access time for all the cells of *Spy_table* verifying: $Index \equiv p \ mod(Total\_number\_of\_sets)$.

3.5. Example and Experimental Results on a Simple Victim

For this section, let us take a schoolbook victim to apply the previously presented covert channel. Our considered victim therefore only performed an addition on 40 bits to simplify the understanding of this toy example. The secret to be extracted was the result of this addition. We now applied the code presented in Figure 3 to this new victim. For demonstration purposes, we gathered 2000 samples for an easier visualization (in practice we need much less samples to extract a secret value). Let us consider that the result of the addition is 672726424737. This corresponded to the binary decomposition: 1001110010100001100111101001110010100001 (40 bits in total). As detailed in the previous section, our covert channel proceeded as follows:

1. The spy code Primed the data cache with any data, as the value itself did not matter. This is agnostic from the victim's behavior and was therefore held for whatever process involving a secret value. Changing the cache's characteristics (number of sets, cache's size, associativity) and the cache's filling policy would require to only change this part of the covert channel;
2. The victim made its addition. The Trojan was run as it was contained in the victim. It decomposed the result of the addition in binary. For each bit, it would, or would not, replace some values inside its array (named *Trojan_table* in Figure 3). The leftmost bit (i.e., at position p = 0) being a 1, the Trojan filled its array at all the following indexes: 0, 256, 512, 768, 1024, 1280, 1536, 1792. All these 8 indexes verified $Index \equiv 0 \ mod$ 256, where 256 was the total number of sets and 0 the position of the value to leak in the binary decomposition. This operation then caused 8 cache misses in the set number 0. The next 1 in the binary decomposition was the bit at position p = 3. The Trojan therefore replaced its array at all indexes verifying $Index \equiv 3 \ mod$ 256. Therefore, it filled the following indexes: 3, 259, 515, 771, 1027, 1283, 1539, and 1795. This caused 8 evictions inside the set number 3 (starting from set number 0). The Trojan repeated these steps for the whole binary decomposition of the secret value to extract;
3. The spy Probed its array and measured the time it took to access every cell. It also generated the logs.

This experiment produced the results given in Figure 5. In the figure, we represented the number of cache misses at every index in *Spy_table* cumulated over the 2000 iterations of the covert channel. We can clearly see 8 patterns corresponding to the 8 ways of the targeted set that represent the Trojan's activity. Those patterns highlight the associativity of the data in the default cache configuration we are studying. The patterns were very similar and carried the same information about the secret to be extracted. In fact, these patterns directly correspond to the evictions done by the Trojan at the cells verifying $Index \equiv p \ mod \ Total\_number\_of\_sets$. Each pattern contained one "peak" corresponding to one of the eight possible values verifying the equation for a given value of p. If we considered the previous example, the Trojan replaced the indexes 3, 259, 515, 771, 1027, 1283, 1539, and 1795 (for $p = 3$). This means that one of the 8 patterns had a peak corresponding to the eviction caused on the index number 3, and another pattern had the peak for the index 259. What is important to note here is that these peaks, however, had the same position inside all of the 8 patterns: the third position. This is because they were all related to an eviction caused in the cache set number 3 corresponding to the fourth bit (i.e., at index number 3) in the binary decomposition of the secret value.

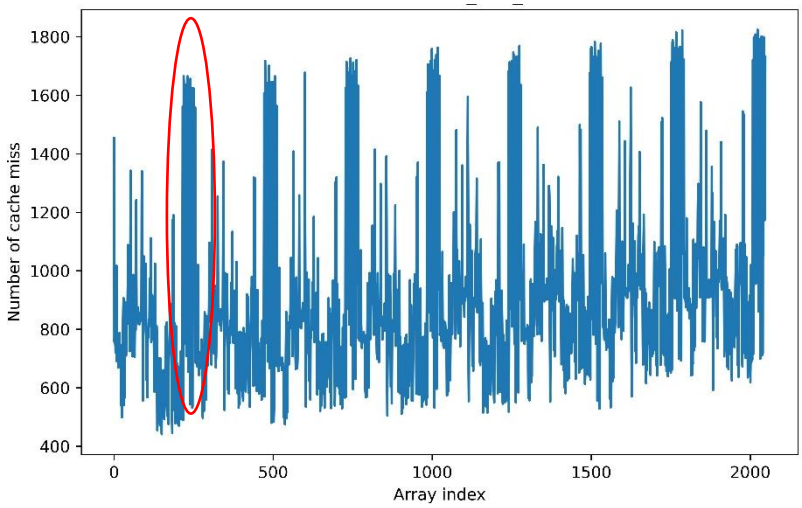

Figure 5. Experimental results for a simple addition victim with the default cache configuration (8-way associative 32 KB writethrough no-write-allocate cache with 256 sets). One of the eight patterns caused by the Trojan's activity has been circled in red for illustration purposes.

Here we wanted to extract 40 bits of secret data; therefore, we targeted the sets number 0 to 39. We noticed the presence of an "offset" at the beginning of the trace, but it did not hinder the analysis in any way. In this figure, every pattern corresponded to the activity of one way inside the cache sets from 0 to 39. For example, the pattern number p represents the amount of cache misses on the way number p inside every cache set from 0 to 39. As we had 8 patterns, we covered all the possible way placements in the figure, as there were 8 ways in total (corresponding to the value of the associativity) in the configuration we study. The data between each pattern corresponded to the amount of cache misses inside the rest of the sets, from 40 to 255. In the sets numbers 40 to 255, the Trojan did not cause any eviction because we only wanted to extract 40 bits therefore we needed only the first 40 cache sets to encode our secret. One can note that we were capable of distinguishing the activity at the cache way granularity here. However, we were forced to cause the Trojan to act at the cache set granularity because of the LFSR. We cannot choose easily and precisely which way the eviction will occur inside a given cache set, as the choice was pseudo-random. Predicting the outcome of the LFSR was possible. However, it implied a longer computation time, thus reducing the overall stealth of the attack.

When we zoomed in on one of the patterns (the circled one in Figure 5), we obtained what is represented in Figure 6. Each pattern was made of 40 values, corresponding to the

40 bits we wanted to extract. A threshold was required to differentiate what we considered as 0 on the trace from what we consider as a 1. We observed that the arithmetic mean of the extreme values of the whole trace (number of cache misses), or $\frac{Max_{value}+Min_{value}}{2}$, worked perfectly for that purpose. Considering the peaks going beyond this value as ones, and the others as zeros, we recovered a value of 1001110010100001100111101001110010100001. This corresponds to the result of the victim's addition; therefore, we recovered the secret value through the CVA6's data cache with a running Linux OS.

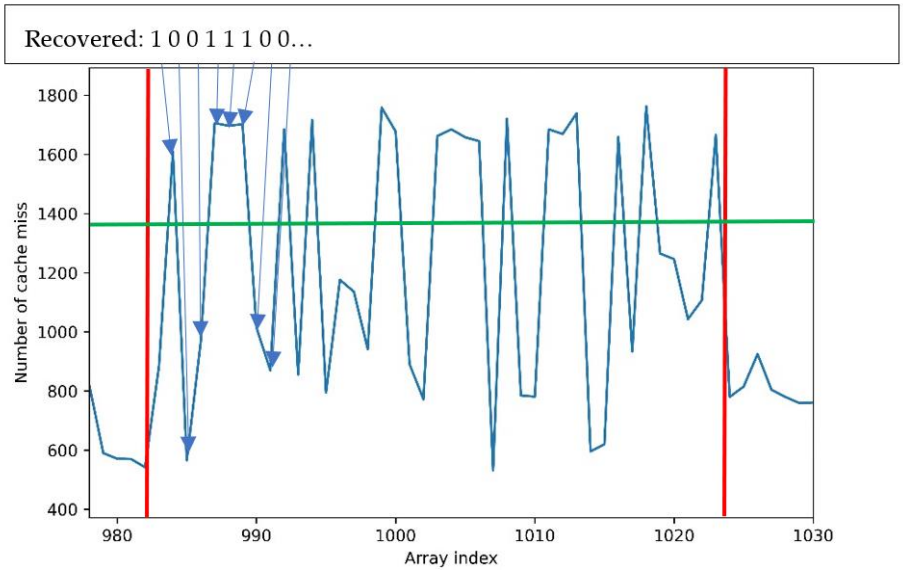

Figure 6. Zoom in on one of the 8 patterns obtained when applying the covert channel on a simple addition victim with the standard data cache configuration. The red bars are delimiting the pattern itself from the rest of the data surrounding it. The green bar is the threshold for considering the resulting peak as a 0 (under the line) or as a 1 (above the line). A few examples of the bit values recovered are given on the upper part of the figure.

4. Applying the Previous Covert Channel in a Realistic Scenario

4.1. Proposed Use-Case: Targeting an Encryption Service

Having studied the covert channel's mechanisms, and a practical example on a very simple victim, let us study a more realistic use case. We proposed to apply our implemented covert channel on an encryption service. Let us consider a victim that is running an AES-based encryption service, in a use-case similar to the one of a Trusted Execution Environment (TEE). Here, the victim proceeds to AES encryptions and decryptions, as requested per the user. These encryptions use a secret key that will be the targeted secret value. As per the previously introduced threat model, the victim can only transmit this key to trusted entities. This means that the user cannot access the secret key, as it is contained in an untrusted domain. Moreover, the user can only interact with the encryption service through defined functions that only give limited outputs: the cipher text for the encryption function, and the plain text for the decryption function. Moreover, we consider that the AES' implementation is based on an open-source code. This means that a Trojan might compromise the AES itself, and its future updates. However, as the compromised AES will run in a given security domain it cannot directly interact outside this domain. Instead of considering a legitimate user, we considered a malicious attacker that was aware of the bug or the Trojan's presence and tried to leak the secret key using our implementation of a micro architectural cache covert channel.

4.2. Experimental Setup and the Challenges to Adapt the Covert Channel to the New Victim

This section describes our setup for this experimentation. The implementation of the covert channel presented in Section 4 is available online on GitHub (https://github.

com/CCALK-work/CCALK (accessed on 4 July 2022)) for reproduction purposes. We chose to work with the "Tiny-AES" [22] open-source implementation of the AES encryption algorithm. We chose a key size of 128 bits. The new victim code in this scenario calls some of the "Tiny-AES" functions when required. We considered that the library itself was compromised and contained the Trojan. For the application use-case, we still considered the application in a TEE environment. Therefore, the attacker (spy) still called the victim process to request for an encryption. The victim then used the library containing the Trojan. It was still possible to carry out the covert channel in the case that the attacker could not directly choose the moment the victim would be run. However, this requires more work for synchronizing the victim and the attacker. As it is not the primary goal of this article, these aspects will not be detailed further.

Changing the victim implied several changes on the covert channel itself. Most of the changes occurred for the Trojan as it was directly related to the type of victim targeted. Each victim required a different Trojan tailored for it. More specifically, the Trojan's code did not change itself. The changes occurred in the interfacing between the Trojan and the target victim code. Compared to our previous experimentations, the Trojan now resided in the "Tiny-AES" library instead of being inserted directly inside the victim, as the secret key we want to extract is computed on inside the library. More precisely, we inserted the Trojan inside the "Key-expansion" function, as it uses the key directly in every AES implementation. In our case the implementation of the "Tiny-AES" requires the victim to manipulate the key directly as it is an input of the library's functions. However, placing the Trojan inside the Key-expansion function should work for any AES implementation as it always uses the key directly. The new pseudo code for the covert channel was presented in Figure 7. The *Prime(Table[])*, *Probe(Table[])* functions did not change at all from the previous pseudo-code presented in Figure 3 and are therefore not presented again. The *Trojan(secret)* function only embeds a supplementary module that will decompose the AES' key in a binary decomposition.

The transition to an AES victim required several code adaptations, mostly on the interface between the new victim and the Trojan. The main challenges that arose from this victim change were: the handling of the Trojan's activation, and the handling of the current part of the key to be extracted in the case it was bigger than the maximum extractable size. Extracting a secret bigger than the maximum chosen extraction size (here we chose 128 bits) required more work but was possible, as detailed later in this section. It was important to note that pattern recognition was harder when approaching of the maximum extraction size for the considered cache configuration (e.g., 256 bits). When extracting secrets of 250 bits and above, the patterns were not separated by enough values to be distinguished by our analysis algorithm. For simplification purposes, we chose to stay with a 128-bit extraction size where the patterns were easier to distinguish.

For the Trojan's activation, we wanted to avoid triggering the Trojan in case a legitimate user (therefore not a malicious attacker) uses the library. This would cause an increase in the computation time and therefore it would make the Trojan easier to spot. To this end, we chose to use a determined sequence inside the input message. For demonstration purposes, we chose that a specific sequence of values for the first 10 8-bit integers (equivalent to the first 80 bits) of the message would activate the Trojan. This sequence was set in the message at the lines 31 to 33 in Figure 7. The probability that a legitimate user triggers the Trojan unwillingly is then 2^{-80}. It was still possible to consider a longer activation sequence to reduce this risk further.

For secrets bigger than 128 bits (the maximum extraction size we chose for easier pattern recognition) we included a selection mechanism. This mechanism is also based on the input message. We chose the eleventh 8-bit integer of the message as an indicator of the part of the secret to be extracted by the Trojan. To extract a bigger secret, we split it into 128-bit parts (for example, as the patterns are easily seen when encoding 128 bits) that we can extract sequentially. For example, if we took a 512-bit secret, we split it into 4 pieces of 128 bits. The Trojan would then look at the eleventh 8-bit integer to know which part

of the secret it had to extract at the current iteration. If this integer was equal to two, the Trojan would then extract the second piece of 128 bits, starting from the bit number 129 of the secret key. From one iteration to another, the spy code incremented these bits in the plaintext. With this technique, we could extract a key up to $128 * 256 = 32{,}768$ bits by pieces of 128 bits. If the secret was bigger than 1024 bits, it was possible to extend the mechanism to two 8-bit integers (the eleventh and twelfth) or more if required.

```
KeyExpension(roundKey, Key)
    ...
    FirstRound()
    ...
    IF (sequence != 0)
        trojan(secret)
    ENDIF
    ...
    OtherRound()
    ...

AES_INITIALISATION(message, key)
    ...
    FOR i in range(0, 10)
        IF (message[i] != i*4)
            sequence = 0
        ENDIF
    ENDFOR
    iteration = message[11]
    ...
    KeyExpension(roundKey, key)
    ...

Victim_encryption(message)
    ...
    AES_INITIALISATION(message, key)
    ...

Spy()
    For i in range(0, NB_ITERATION)
        FOR k in range(0, 10)
            message[k] = k*4
        message[11] = i
        ENDFOR
        For j in range(0, NB_SAMPLES)
            Prime(spy_table);
            Victim_encryption(message);
            Probe(spy_table);
        ENDFOR
        Generate_logs();
    ENDFOR
```

Figure 7. Pseudocode for the covert channel targeting a "Tiny-AES" implementation of the AES encryption algorithm.

In practice, our implementation of the Trojan code represented 40 lines of code inserted inside the "Tiny-AES" library that was composed of approximately 570 lines of code. This meant that our Trojan increased the size of the library's code by around 7%. The Trojan's code should remain the same for bigger libraries, encompassing several other algorithms for instance. As the "Tiny-AES" had a small code size, the Trojan would probably be stealthier when placed in bigger libraries. Moreover, it was possible to improve the stealth of our Trojan's implementation, and to reduce its number of lines. However, it was not the goal of our experimentations, and we did not go any further regarding stealth. This part was left for future work.

4.3. Experimental Results and Limitations

When we applied out implementation of a micro architectural covert channel on a "Tiny-AES" implementation having a 128-bit key, we obtained the results given in Figure 8. We chose to run the covert channel with 2000 samples to improve the accuracy of the key's

retrieval. The trace was generated by our Python script using the logs produced by the spy code. The logs contain the time it took the spy to access each cell of *Spy_table*.

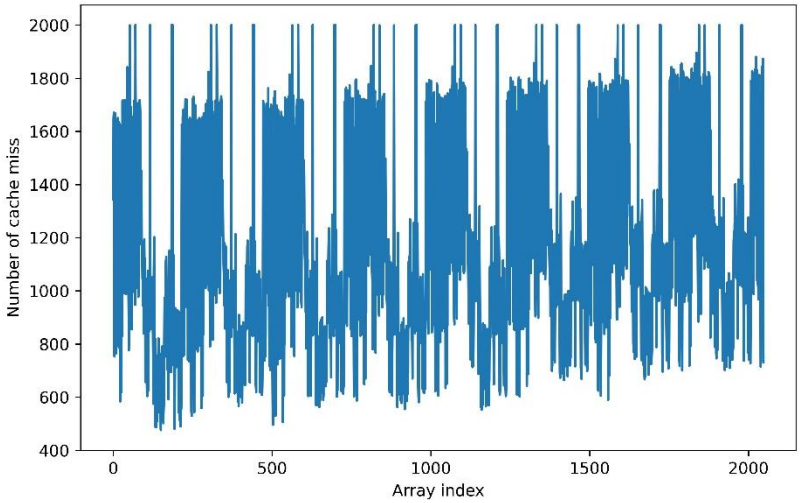

Figure 8. Trace obtained when carrying out our covert channel on the "Tiny-AES" implementation of the AES with a 128-bit key and 2000 samples.

Analogously to the traces obtained for the simple addition victim, we can observe the same type of patterns on this trace. As we were still working with the default cache configuration, we still had 8 distinct patterns that looked very similar, as expected. Here we could also clearly see the offset's presence that split one pattern over the figure. We chose not to correct this offset, as it did not influence the success of the attack at all. The result of zooming on one of the patterns is given in Figure 9.

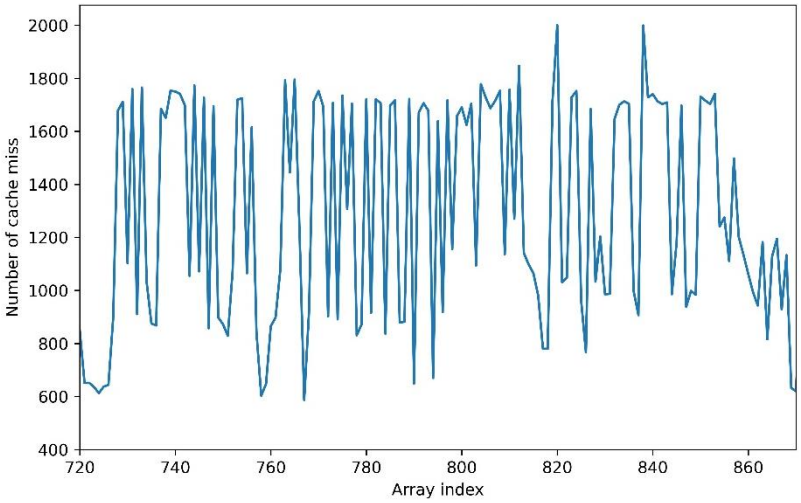

Figure 9. Trace obtained when zooming on one of the 8 patterns visible on the full trace of the covert channel targeting an AES with 2000 samples.

Once we generated the trace given in Figure 8, our Python script could recognize the patterns. As for the simple victim, the script picked the first complete pattern it finds and set the threshold. It then recovered a binary value. Here, the value recovered was composed of 128 bits just like the secret key. We then compared the value recovered to the binary decomposition of the key to check the attack's success.

We achieved in recovering the 128-bit key with a success rate of 97.6%. More precisely, we were able to recover 97.6% of the secret key for every covert channel we carried out. The 2.4% error was due to the presence of ways in the cache where there was an important

amount of cache misses, independently of the Trojan's activity. This was visible with the presence of some very high peaks outside of any recognizable pattern (some are also located inside the patterns). This meant that there was some activity that was not caused by the Trojan nor the spy that evicted the spy's data before it could access it again. The work proposed in [23] carried out a Prime + Probe covert channel on CVA6 in a BareMetal simulation environment. There was no specific mention of such perturbations except for what the authors named the "CPU deadzone". The CPU deadzone consists in a cache area being constantly filled by the CPU when computing. This zone typically contains the running programs' addresses, intermediate values, etc. It has been named "deadzone" because no data can be kept in it for longer than a few cycles as the cache content is constantly replaced during calculations. In our case, this deadzone does not seem to be the cause of the phenomenon observed as the indexes affected inside the cache do not match the author's observations. Moreover, the number of perturbations observed highly differs in the case of a running OS. Approximately 8 cache sets are affected by the perturbations in the case of an OS compared to more than 50 for the CPU deadzone. This effect is therefore specifically due to the presence of the Linux OS.

We interpreted these peaks as the processes related to the OS that are running while we carry out our attack. However, this could not be demonstrated. We could not ensure that our covert channel (victim + Trojan + spy) can be run within the timeframe allocated by the OS. Indeed, some other processes are run in between, after the Prime step, but before the Probe step. This causes some of the spy's data to be evicted. These unwanted peaks are located at some indexes inside the patterns, causing 2.4% of the key being recovered incorrectly upon every extraction. Increasing the sample count reduces the impact of these unwanted peaks, and thus increases the success rate of the covert channel. To illustrate, the covert channel can recover 95% of the key correctly using 10 samples only. This considerably reduces the computation time for a slight decrease in the success rate.

Moreover, the covert channel take approximately 1 s when using 10 samples. We measured that we could run about 40 AES per second on our FPGA board without any covert channel. When using 10 samples, this corresponds to 10 AES encryptions per second, resulting in a noticeable slowdown. Again, we did not focus on making the covert channel stealthy, and future work could focus on this aspect.

4.4. The Impact of the Cache Architecture and Perturbations on the Covert Channel

We carried out some other experimentations focusing on the impacts of the "environment" on the covert channel efficiency. We consider that the "environment" is mainly composed of the data cache's architecture, and the activity of the OS. These two elements are the ones having the highest influence on our covert channel.

For the cache architecture, we carried out our covert channel with several changes in the cache's parameters. Modifying the cache size implied that we needed to adjust our arrays to fit the new dimensions. This did not impact the success rate, however. We also tried to modify the LFSR to see if it impacts the outcomes of the covert channel. We alternatively changed the polynomial used while keeping the same degree, and changed the degree. In both cases, we did not notice any modification in the attack's behavior. The success rate did not evolve significantly. However, the cache associativity had a very high impact on the cover-channel's behavior. Changing the associativity means changing the amount of values we can extract. The higher the associativity, the more ways each set would contain. For a fixed cache size, this meant that increasing the associativity decreased the number of sets, and thus the amount of bits a Trojan can transmit. We had an increased number of patterns on our traces, but each pattern was composed of fewer values. The opposite was also true: decreasing associativity with a fixed cache size increased the number of sets and thus the number of bits that can be extracted. We could still adapt the attack in several different scenarios. For associativity values of 4, 8, 16, and 32, the covert channel was still working, provided it was adapted to fit the new environment. The success rate in these cases did not evolve significantly. We could not try some extreme

cases, such as changing to a direct-mapped cache, or a fully associative cache (meaning that we have only 1 set composed of as many ways as the cache size requires it) because these configurations are not supported by the CVA6's data cache. As a conclusion, we can say that the cache structure has an impact on the covert channel. However, a simple adaptation inside the code is sufficient. Cache associativity has more impact on the attack as it directly modifies the amount of information transmitted. Intuitively, and considering the previous results, a fully associative cache would be more resilient to our proposed covert channel, as we would only be able to extract one bit at a time (as there is only one cache set). This would not make the attack impossible, but less practical and slower.

In an effort to propose an even more realistic approach to the application of our covert channel, we carried it out in a "noisy" environment. We added some genuine clients (e.g., also embedding the Trojan but not activating it) running in parallel with a malicious user. We placed two other clients using the same encryption service as the attacker, with the same execution priority. These clients are doing AES encryptions in tight and infinite loops, in order to maximize the perturbations caused. These perturbations were created to amplify the effect limiting the success rate because of the unwanted peaks appearing in the patterns. The effects caused by these two genuine clients were observable. They caused a significant drop in the success rate. This means that we had to increase the amount of samples to achieve a success rate of 95%. For example, the covert channel without perturbations can achieve 95% of success rate with only 6 samples. To achieve 95% of success rate with the perturbations, we needed to use 10 samples instead. Less samples produced a degraded success rate with the perturbations. Similarly, when adding 3 genuine clients, 15 samples were needed to achieve 95% of extraction rate. For 4 processes it goes up to 22 samples. The observed trend and the precise number of samples vary with the type of code implied and the attack's implementation itself. To conclude, if some processes are running concurrently with our covert channel, the perturbations degrade the quality of the extraction, but it is still possible to account for it by increasing the amount of samples considered.

5. Conclusions and Perspectives

We presented an implementation of an access-driven cache-based micro architectural covert channel on the RISCV CVA6 core, in a running OS context. This attack is practical and has been applied to a simple victim consisting of an addition process then to a more realistic use-case, targeting an implementation of the AES encryption algorithm, with a 128-bit key. The attack's success rate is around 95% for 10 samples.

We also studied some of the challenges that the presence of an OS imply when carrying out such a covert channel, such as the presence of other processes running concurrently, or aspects about the scheduling that have to be considered when developing a covert channel attack. Moreover, we also showed that the cache's architecture and dimensioning has an impact on the ability of these covert channels to extract information, and therefore on the cache's level of vulnerability to such threats.

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. This paper proposes a study of the mechanisms involved in a cache covert channel and their limitations in an effort to help the development of future mitigations against micro architectural covert channels. These results are applied to a realistic scenario with a cryptographic implementation to show these threats are practical even though they require a specific methodology to be adapted from a given processor to another.

A future work is to propose a stealthier version of the Trojan's implementation to make the covert channel even more realistic and practical. It is also important to work on the potential mitigations to such covert channels, as none of the observed perturbations could completely prohibit the extraction.

Author Contributions: Conceptualization, all authors; methodology, all authors; software, all authors; validation, all authors; formal analysis, all authors; investigation, all authors; resources, all authors; data curation, V.M. and E.T.; writing—original draft preparation, V.M.; writing—review and editing, all authors; visualization, all authors; supervision, Y.T. and R.L.; project administration, Y.T. and R.L.; funding acquisition, Y.T. and R.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Van Bulck, J.; Minkin, M.; Weisse, O.; Genkin, D.; Kasikci, B.; Piessens, F.; Silberstein, M.; Wenisch, T.F.; Yarom, Y.; Strackx, R. *Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution*; USENIX Association: Berkeley, CA, USA, 2018.
2. Canella, C.; Genkin, D.; Giner, L.; Gruss, D.; Lipp, M.; Minkin, M.; Moghimi, D.; Piessens, F.; Schwarz, M.; Sunar, B.; et al. Fallout: Reading kernel writes from user space. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019. [CrossRef]
3. Schwarz, M.; Lipp, M.; Moghimi, D.; Van Bulck, J.; Stecklina, J.; Prescher, T.; Gruss, D. ZombieLoad: Cross-privilege-boundary data sampling. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019. [CrossRef]
4. Van Bulck, J.; Moghimi, D.; Schwarz, M.; Lippi, M.; Minkin, M.; Genkin, D.; Yarom, Y.; Sunar, B.; Gruss, D.; Piessens, F. Lvi: Hijacking transient execution through microarchitectural load value injection. In Proceedings of the 41th IEEE Symp. on Security and Privacy, San Francisco, CA, USA, 18–21 May 2020; pp. 1399–1417. [CrossRef]
5. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; et al. Meltdown: Reading kernel memory from user space. In Proceedings of the USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018.
6. Allaf, Z.; Adda, M.; Gegov, A. A comparison study on flush+reload and prime+probe attacks on AES using machine learning approaches. In *UK Workshop on Computational Intelligence*; Springer: Cham, Switzerland, 2017; pp. 203–213.
7. Yuval, Y.; Katrina, F. *Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack*; USENIX Association: Berkeley, CA, USA, 2014.
8. Gruss, D.; Maurice, C.; Wagner, K.; Mangard, S. Flush+flush: A fast and stealthy cache attack. In Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, San Sebastián, Spain, 7–8 July 2016; Volume 9721, pp. 279–299.
9. Disselkoen, C.; Kohlbrenner, D.; Porter, L.; Tullsen, D. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; Volume 17, pp. 51–67.
10. Wistoff, N.; Schneider, M.; Gürkaynak, F.K.; Benini, L.; Heiser, G. Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V Core. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 627–632. [CrossRef]
11. Wistoff, N.; Schneider, M.; Gürkaynak, F.K.; Heiser, G.; Benini, L. Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning. *IEEE Trans. Comput.* **2022**, 1–11. [CrossRef]
12. The seL4 Microkernel. seL4 Foundation. Available online: https://sel4.systems/ (accessed on 16 December 2022).
13. Zaruba, F.; Benini, L. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Trans. VLSI Syst.* **2019**, *27*, 2629–2640. [CrossRef]
14. RISC-V International. Available online: https://riscv.org (accessed on 4 December 2022).
15. CVA6 Core. ETH Zurich. Available online: https://github.com/openhwgroup/cva6 (accessed on 27 June 2022).
16. Martinoli, V.; Bouagoun, A.; Leveugle, R.; Teglia, Y. CVA6′s Data Cache: Structure and Behavior. 8 February 2022. Available online: https://arxiv.org/abs/2202.03749 (accessed on 27 June 2022).
17. Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. JSOF. Available online: https://www.jsof-tech.com/ripple20/ (accessed on 17 September 2020).
18. Digilent reference—Genesys 2. Digilent. Available online: https://digilent.com/reference/programmable-logic/genesys-2/start (accessed on 27 June 2022).
19. Xilinx Vivado. Xilinx. Available online: https://www.xilinx.com/products/design-tools/vivado.html (accessed on 27 June 2022).
20. CVA6-SDK. Open Hardware Group. Available online: https://github.com/openhwgroup/cva6-sdk (accessed on 27 June 2022).
21. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, 2017. Available online: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (accessed on 5 December 2022).

22. Tiny-AES-c. Available online: https://github.com/kokke/tiny-AES-c (accessed on 27 June 2022).
23. Martinoli, V.; Teglia, Y.; Bouagoun, A.; Leveugle, R. Recovering Information on the CVA6 RISC-V CPU with a Baremetal Micro-Architectural Covert Channel. In Proceedings of the 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 12–14 September 2022; pp. 1–6. [CrossRef]