REHAD: Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection

Yuxiao Mao, Vincent Migliore, Vincent Nicomette

REHAD: Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection

Yuxiao Mao, Vincent Migliore, Vincent Nicomette
LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
Email: yuxiao.mao@laas.fr, vincent.migliore@laas.fr, vincent.nicomette@laas.fr

***Abstract*—Cache side-channel attacks consist, for a malicious process, to infer the current state of the cache by measuring the time it takes to access the memory, and indirectly gain knowledge about other processes sharing this same physical cache. Because cache side-channel attacks leverage a hardware leakage without requiring any physical access to the devices, they represent very serious threats. Among the runtime detection techniques for cache side-channel attacks, hardware solutions are usually fine-grained and benefit from less performance overhead than software solutions. However, they are not flexible enough to suit the rapid evolution and appearance of software attacks. In this paper we describe REHAD, a novel attack detection architecture that uses reconfigurable hardware. More precisely, it includes a hardware detection module that can be reconfigured by means of a trusted software kernel, to adapt to the level of threats and to detect new attacks. This architecture also benefits from hardware parallelism to fill the frequency gap between reconfigurable hardware and core processor. REHAD has been integrated into the ORCA softcore RISC-V on a FPGA and two common cache side-channel attacks have been successfully detected.**

***Index Terms*—Reconfigurable architectures, intrusion detection, microarchitectural timing attacks, RISC-V**

1. Introduction

In recent years, more and more attention has been paid to information leakage at software/hardware interfaces. During the execution of software, the hardware handles some information related to the execution. This information can be storage information, such as the Branch Target Buffer (BTB) that memorizes the jump history; it can also be timing information, such as the waiting time after requiring a busy peripheral. Even if the running software itself has no vulnerabilities, this information can still be leaked through side-channels such as timing channels. A malicious process may infer the internal state of the hardware by measuring the time taken to execute some specific instructions, and obtain information leaked by the victim process sharing the same hardware, violating by this way the inter-process isolation. Such attacks are so-called microarchitectural timing attacks [1].

Cache Side-Channel Attacks (CSCAs) are microarchitectural timing attacks that specifically target the shared caches. A malicious process can use CSCAs to deduce memory access sequences of the victim, including instructions sequences and data sequences. In this way, the attacker can learn information about the program control flow or secret keys, etc. For example, the Last-Level Cache (LLC) is shared by all processor cores, and may be used by a malicious process to target any victim process running on the processor. CSCAs can also be used as a tool for other attacks, such as Spectre [2] and Meltdown [3], to read the information leaked by the victim process during transient execution. As CSCAs do not require physical access, and do not rely on any vulnerabilities in the victim process, they are powerful attacks.

Defense mechanisms against CSCAs can be divided into two categories: prevention and detection [4]. Prevention at hardware level essentially consists in clock modifications and redesign of the shared hardware architecture. Prevention at software level consists in enhancing time and space isolation. Software developers may also apply constant time programming for critical code section. As regards to detection, the mechanisms may either be static (analysis of binary files), or dynamic (runtime monitoring of process activities in order to detect malicious behavior).

In this paper, we focus on dynamic monitoring. Such monitoring technique offers several advantages: 1) as it is performed at runtime, it allows the operating system to dynamically activate or not some defense mechanisms (that may be costly), according to the outputs of the monitoring, and 2) it has the potential to detect unknown attacks. Software-based monitoring can be quickly deployed and easily updated. However, the hardware internal state used by CSCAs is not directly visible by the software; hence software-based monitoring must only rely on accessible hardware resources, which limits its efficiency. Traditional hardwired hardware-based monitoring has low overhead and is fine-grained, but lacks flexibility to respond to the variety and the rapid evolution of software attacks.

In this paper, we introduce the REHAD (REconfigurable Hardware for Attacks Detection) architecture. Our contributions are:

- A highly flexible hardware/software co-design attack detection architecture, with a hardware *detection module* made up of reconfigurable hardware and communicating with the core processor;
- An adaptative hardware parallelism to fill the frequency gap between the *detection module* and the core processor, and the definition of the communication interface between them;
- The design of simple heuristics, implemented in the *trusted software kernel* running on top of the core processor, that can efficiently detect CSCAs,

by performing simple pattern matching on the information provided by the *detection module*;
- The implementation of a proof-of-concept on a Field-Programmable Gate Array (FPGA), based on the Orca RISC-V softcore processor [5]. This experiment allowed us to successfully detect two common CSCAs (Flush+Reload and Prime+Probe) and shows the relevance of attack detection with low-frequency reconfigurable hardware.

Section II presents CSCAs attacks and describes some research works that focus on 1) CSCAs detection and 2) the use of reconfigurable hardware with processors. Section III provides an overview of the REHAD architecture as well as the different components of the architecture. Section IV is dedicated to the presentation of an experiment, showing the relevance of our architecture and Section V concludes and discusses future work.

2. Background and State of the Art

2.1. Cache side-channel attacks

Modern processors run at a high-frequency and memory accesses often require hundreds of processor clock cycles to be executed. In order to reduce the processor's waiting time to fetch instructions and data from main memory, a memory cache is used. When the processor wants to access some data, this data as well as the data nearby are first loaded into the cache for later use. Therefore, the time required to read an information when it is in the cache (cache hit) or not (cache miss) is different. This timing difference is the leakage used in CSCAs.

The cache is a faster and smaller structure compared to the main memory, and can be implemented with different strategies (direct-mapped, fully associative and set associative). In direct-mapped cache strategy, a given memory address can only be present in a single cache line whereas it can be present in all cache lines in fully associative cache. The set associative strategy is an intermediary strategy in which a given memory address can be present in specific set of lines (generally between 2 and 24).

Two common CSCAs strategies exist. The **Flush+Reload** [6] strategy requires a shared memory between the attacker and the victim, such as shared libraries or deduplicated memory pages. The attacker first flushes a line of the shared memory off the cache, and waits for the victim process to run. The attacker then measures the time it takes to access this memory line, to determine whether this line has been used or not by the victim. The **Prime+Probe** [7] strategy targets set associative caches and does not require any shared memory between the target and the victim processes. The attacker first primes a cache set with his own data, which ensures that the victim has no cache line in the given cache set, and waits for the victim process to execute. The attacker then probes the overall access time to the data previously filled by himself, to identify whether one or more cache lines in this set has been used or not by the victim.

```
1    rdtscp              ; get timer value
2    mov %eax, %esi
3    mov (%r9), %eax     ; 1 memory access
4    rdtscp              ; get timer value
5    sub %esi, %eax      ; compute difference
6    ...                 ; other instructions
7    clflush (%r9)       ; flush memory line
```

Listing 1. Time measurement logic of Flush+Reload attack on x86

```
1    rdtscp              ; get timer value
2    mov %eax,%r10d
3    mov (%rdi),%rax     ; 8 memory access
4    mov (%rax),%rax     ; using pointer-chasing
5    mov (%rax),%rax
6    mov (%rax),%rax
7    mov (%rax),%rax
8    mov (%rax),%rax
9    mov (%rax),%rax
10   mov (%rax),%rdi
11   rdtscp              ; get timer value
12   sub %r10d,%eax      ; compute difference
```

Listing 2. Time measurement logic of Prime+Probe attack on x86

For better accuracy, these attacks try to use as few instructions as possible between time measurements. In Mastik toolkit [8], the measurement is performed for only one memory access instruction in Flush+Reload attack, as shown in Listing 1, and 8 memory access instructions in a Prime+Probe attack targeting a set of size 8, as shown in Listing 2. Our detection logic is based on this feature of CSCAs.

2.2. Cache side-channel attacks detection

CSCAs detection mechanisms can be divided into two categories: static analysis and runtime monitoring.

Static analysis aims at analyzing a binary file before its execution [9], [10]. The instructions of the file are analyzed in order to identify whether it contains a CSCA or not. A suspicious program may not be allowed to run, or may be allowed to only run under strict conditions. This method is suitable for scans in the applications store, for example. However, when the attack payload is protected by encryption, obfuscation or dynamic loading, static analysis is not efficient anymore.

Runtime monitoring may be performed with or without hardware modification. The monitoring techniques without hardware modification simply read information provided by the existing hardware, and take a decision with software logic. The existing hardware may be the Hardware Performance Counters (HPCs) for instance, present in most modern processors. The software logic then uses thresholds, heuristics, or machine learning algorithms to identify CSCAs [11].

Thanks to hardware modification, it is possible to collect more precise and useful information for attack detection. Chen et al. [12] propose to modify the structure of the cache and other shared hardware, in order to include custom counters, that they use instead of HPCs. This approach is interesting but it requires some modification of every hardware component shared between the processes that are supposed to be protected, which is costly and hard to maintain.

Ozsoy et al. [13] propose malware-aware processors for malware detection. They extract information directly from the processor, and analyze this information using hardware-implemented machine learning. The authors mention that the thresholds used in machine learning

methods can be configurable. This work has some similarities with REHAD, but is limited to a single analysis algorithm, and puts aside the possibilities to improve and replace the algorithm in the future. The main contribution of REHAD is to ensure the flexibility of the *detection module* by using reconfigurable hardware.

2.3. Reconfigurable hardware

Reconfigurable hardware, such as FPGAs, is hardware that can be reconfigured by the user after fabrication, so that it can be implemented with any arbitrary logic. For that purpose, FPGAs offer numerous logic, routing and memory resources to the user. Taking into account this high level of flexibility, FPGAs usually require large circuits and suffer from much lower frequency than hardwired implementation for the same logic [14].

Using reconfigurable hardware along with hardwired processors is not a new research topic [15]. Reconfigurable hardware benefits from highly parallel execution capabilities to speed up the processor's calculations, and can be reconfigured to implement different algorithms. It has been successfully used in many fields such as image processing and communication. Regarding the security domain, reconfigurable hardware has been proposed for cryptography acceleration and secret protection, for power and communication monitoring against hardware attacks [16]. However, to the best of our knowledge, no research work has proposed the use of reconfigurable hardware to monitor the running software on a processor for CSCA detection.

3. REHAD Architecture

3.1. Overall architecture

The REHAD architecture is shown in Fig. 1. This architecture is composed of a main processor core, a *detection module* made up of reconfigurable hardware, interconnected by three *communication channels* made up of static hardware, and a *trusted software kernel* located in the processor. The *detection module* aims to analyze data provided by the processor core in real-time, and provides hardware relevant information to the *trusted software kernel* for further decision. Furthermore, the *detection module* can be reconfigured to adapt to new threats or attacks.

Our architecture contains two clock domains. The processor core runs at high-frequency whereas the *detection module* runs at a lower frequency due to its flexibility. As a consequence, the three *communications channels* (made up of static hardware) between the processor core and the *detection module* must perform clock domain crossing. The *detection module* uses hardware parallelism to process simultaneously in one single (slow) clock cycle multiple data from multiple (fast) clock cycles of processor. For example, a *detection module* running at 200 MHz, and processing 16 information data at the same time, can support a modern processor with a clock frequency of 3.2 GHz. This level of parallelism and these execution frequencies sound reasonable.

For embedded devices that run at lower frequency, a serial detection logic can be considered because parallelism is not required anymore (the core processor and FPGA run at the same frequency clock). However, our parallel design is still relevant because a low-frequency *detection module* exhibits price and energy consumption advantages.

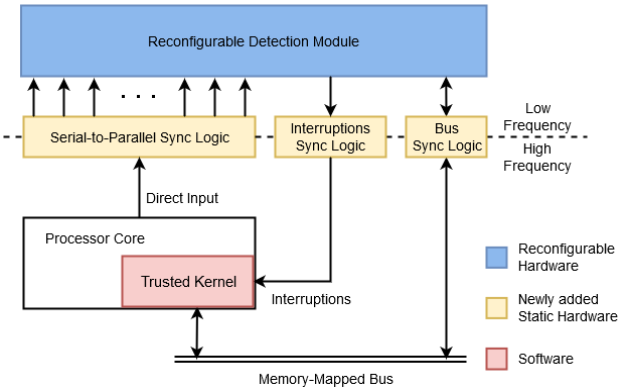

Figure 1. Overall design of REHAD, a highly flexible hardware / software co-design attack detection architecture, with a hardware *detection module* made up of reconfigurable hardware.

3.2. Communication channels

The communication between the processor and the *detection module* goes through three clock domain crossing *channels*. The first *channel* is a serial-to-parallel circuit and is the most important *channel* for runtime monitoring. The high-frequency serial data are buffered and converted into low-frequency parallel data. It is used to transfer information (such as the instruction flow) extracted from the processor to *detection module* for further analysis. The second *channel* is an interruption mechanism used to notify the processor of any urgent abnormal situation. This *channel* can be disabled to avoid penalizing the normal execution of programs too much, and replaced by a periodic polling mechanism from the *kernel* using the third *channel*. This third *channel* is a memory-mapped bus, which allows the *software kernel* to access the *detection module* as if it was a peripheral. Various actions such as counter reading, thresholds setting and requests of reconfiguration are possible through this channel.

Since processor microarchitecture and *communication channels* cannot be modified during their entire lifetime, data extracted from the processor and sent to the detection module through the first *channel* must be chosen wisely. For Flush+Reload and Prime+Probe attacks, short sequences with frequent time measurement instructions, as shown in Listing 1 and 2, must be repeated. Thus, the observation of the instructions executed by the processor appears as a good choice to efficiently detect CSCAs attacks. Let us note that the observation of the instructions executed may also help to take into account the hardware impact of transient execution used by Spectre attack for instance. In other words, the choice of using instructions as input for the CSCAs detection increases the likelihood of detecting other software-based attacks that exhibit specific instruction patterns.

3.3. Trusted software kernel

As hardware-level events alone are not sufficient to take complex decisions, a *trusted software kernel* is included in REHAD architecture. It may be embedded in the operating system kernel itself or an hypervisor running on the main processor and communicating with the *detection module*. This *kernel* plays multiple roles in REHAD.

During runtime detection, the *kernel* receives interruptions and data such as counter values from the *detection module*. It synthesizes the situation with other known information about the running process and the execution environment, figures out whether the process is suspicious or not and its corresponding threat level. When a suspicious process is found, the *kernel* can decide to apply suitable countermeasures according to the threat level and security policy, such as killing the process or forcing its migration into a completely isolated environment.

The security requirements may vary according to the software running on the system. For example, when an encryption process is running, the system needs higher detection sensibility to prevent any possible leak of secret; when a benchmark process is running, the system may choose to lower the security level to deal with unwanted false positives. The *kernel* can provide thresholds and configuration parameters to the *detection module* through the memory-mapped bus or even decides to reconfigure it with a special logic suited for one attack type. This allows to keep the *detection module* up to date when new attacks emerge, as long as the adequate hardware redesign is available.

3.4. Hardware detection logic against cache side-channel attacks

Hardware monitoring is only designed to run at the same frequency as the main processor. We want to take full advantage of the parallel nature of the hardware to maximize out detection capability in this parallel input context.

The *detection module* takes the executed instructions as input, as mentioned in Section 3.2. Fig. 2 illustrates the situation in which the number of parallel inputs is 4, and the attack pattern to be detected corresponds to two relatively close instructions getting the value of the internal timer of the processor (such as `rdtscp` instruction in Listing 1).

First, each instruction executed on the core processor is analyzed in order to identify whether it is related to the timer manipulation or not. As the clock rate of the *detection module* is four times slower than the clock rate of the processor in this example, four instructions may be executed during one clock cycle of the *detection module*. If one of these instructions manipulates the timer (so-called a "timer instruction"), a bit is set in a specific register (so-called *detection register*) dedicated to the current clock cycle of the *detection module*. Each bit of this *detection register* indicates the presence or not of instructions of some specific category during the current cycle. Each bit is set to 0 or 1 if at least an instruction of this category has been executed or not during the current cycle. Let us note that we do not count the exact number of occurrences of each instruction because it may slow

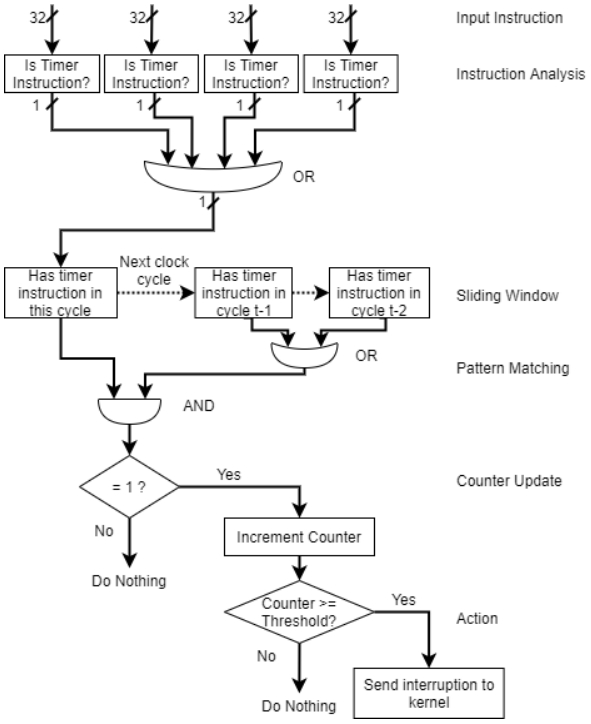

Figure 2. One possible design of *detection module* in REHAD. It takes 4 32-bit instructions as input, converts them to serial information about instruction, and uses simple pattern matching to detect attacks.

down the *detection module*. As for now, for our current experiments, the *detection register* includes only two bits, i.e., two categories of instructions (one bit for the timer instructions and one bit for the instructions that flush the cache). The *detection registers* corresponding to the three latest clock cycles are stored to be used for the detection. Then the detection logic consists in finding a specific pattern using these three *detection registers* in a sliding window strategy. In this example, the pattern corresponds to the presence of at least one timer instruction in the current clock cycle (i.e., the bit corresponding to the timer instruction is set to 1 in the current *detection register*) and a second timer instruction in one of the previous two clock cycles. If such a pattern is identified, a dedicated attack pattern counter is incremented. When the counter exceeds a specific threshold set by *software kernel* and the hardware interruption is enabled, the *detection module* interrupts the processor to warn of an abnormal situation.

This detection logic converts parallel instructions to a simple serial form, loosing information such as instruction execution order and multiple appearance of the same instruction into a single cycle. Nevertheless, this method is still precise enough to detect the presence of CSCAs. As this part of logic is fully reconfigurable, some other detection logic, such as machine learning algorithms specifically designed, can also be used instead.

4. REHAD implementation for Flush+Reload and Prime+Probe detection

This section describes an experiment that was carried out in order to assess the relevance of our architecture. First, we present the implementation details of the

different components of our proof of concept, then we describe the Flush+Reload and Prime+Probe experiments we performed. Finally, we present the result of our implementations.

4.1. Hardware and software configuration

This implementation was carried out on a Xilinx ML605 Evaluation Board (including a Xilinx Virtex-6 FPGA) using the Orca RISC-V softcore processor as processor cores, configured with the RV32IM standard of RISC-V Instruction Set Architecture (ISA): 32-bit instruction size, with extension for integer multiplication and division. It is an in-order processor with separate direct-mapped data cache and instruction cache. Each cache is 512-byte size, divided into 16 cache lines of 32 bytes. It has two memory-mapped bus interfaces, a cached one with an AXI4 interface connected to a 32 KB main memory implemented using one chip Block RAMs and an uncached one with an AXI4Lite interface connected to *detection module*. It also includes a Universal Asynchronous Receiver-Transmitter (UART) that we can use to send commands (as illustrated in the next section). We modified the processor to observe the instruction transmitted from pipeline decode stage to the execute stage, and 1-bit indicating the end of execution of the given instruction.

We modeled the frequency difference between the processor and the *reconfigurable detection module* with a 1/16 clock divider. We coded the three *communication channels* and the *detection module* in VHDL language. The *detection module* is placed into a dedicated area that can be partially reconfigured in FPGA, to simulate the situation where the *detection module* can be reconfigured while other hardware remains unmodified.

Regarding the software logic, we used a bare metal loop to run different processes one after the other. For now, the *kernel* is only responsible for reading information from the *detection module*, setting thresholds, and reacting to interruptions.

We implemented Flush+Reload attack and Prime+Probe attack on RISC-V ISA architecture by adapting the code of the Mastik toolkit which includes exploits coded for the x86 ISA architecture. In RISC-V ISA, the instructions that are able to supply an accurate time value are instructions that access the processor internal timer, such as `rdtime`. The cache flush instruction has not yet an official definition, but we found its implementation in Orca by analyzing its code. As such, in our adapted version of attacks, the `rdtime` instruction was used instead of the `rdtscp` instruction of x86 ISA, and the processor-specific flush instruction was used instead of `clflush`.

4.2. Flush+Reload detection

The Flush+Reload attack experiment was performed by implementing a malicious process and a victim process running on top of the *kernel*. The scenario is the following one: The malicious process executes Flush+Reload instructions on 8 memory addresses (shared with the victim process) associated with different cache lines, at each execution slot. At the same time, the victim process performs one access to a memory address as indicated by the command sent from the UART. In this architecture, when data in a given memory address is in the cache, the access time measured for this address is 2 processor cycles; when it is in the main memory, the access time measured is 20 processor cycles due to time used to fill one cache line. As a consequence, when the attacker performs the Flush+Reload attack, two timer instructions that are either 2 or 20 processor clock cycles away are executed. From the point of view of the *detection module*, this means that the bit related to the timer instruction is set to 1 in either 1 or 2 of the three *detection registers* (1 clock cycle of the *detection module* corresponds to 16 cycles of the core processor). We chose the following pattern to detect the attack: "an attack is detected if 2 out of 3 *detection registers* contain at least one timer instruction". Let us note that this can ensure that the 20 cycles cache miss event is detected but that the 2 cycles cache hit may be not (in the case where the two timer instructions are executed within the same *detection module* clock cycle). However, as the attack both requires cache miss and cache hit to be successful, we are actually able to detect it.

Let us note that, in our experiment, the malicious process can only detect the victim accesses to 5 out of 8 lines of monitored memory. This limitation is due to the fact that malicious process also needs to use the data cache for its normal execution, that it uses the monitored cache line and influences the monitored result. We counted the number of slow cycles during which at least one cache flush instruction was executed, and the number of timer instructions that occur twice in three slow cycles. Both approaches were able to detect the presence of Flush+Reload attacks.

4.3. Prime+Probe detection

As cache size is small in our architecture, our Prime+Probe attack is adapted from a version that targets L1 data cache, which probes all sets in a cache. To have a more realistic Prime+Probe attack, we emulate a 2-Way associative cache from the attacker perspective. It is convenient from our 16 line direct mapped cache by accessing lines with a step of 8 line instead of 16. The malicious process performs Prime+Probe for each set, at each execution slot. An analysis of the assembly code shows that there were 18 instructions between the two timer instructions, Two of them are memory access instruction, and the timing difference measured is between 69 cycles and 112 cycles. With a threshold appropriately chosen, the malicious process can detect victim access to 5 out of 8 sets.

The attacker evicts the cache by priming with his own data and does not need to execute any flush instruction. The distance between two timer instructions is longer than in Flush+Reload. As a consequence, the aforementioned attack detection designed for Flush+Reload becomes ineffective. We increased the size of the sliding window to include 8 *detection registers* and reconfigured the *detection module*, this permitted the detection of the presence of Prime+Probe attack.

In comparison with our implementation for Flush+Reload attack detection, we use the exactly same hardware, with just a different configuration of

the *detection module*. The reconfiguration process does not require any hardware replacement and is fully controlled by *trusted software kernel*. To sump up, the use of reconfiguration hardware in REHAD architecture makes hardware-based detection of Prime+Probe attacks possible.

4.4. Implementation results

The hardware resource utilization and detection capability of the two configurations of *detection module* are summarized in Table 1. Both configurations are based on the same REHAD implementation with clock division rate set to 1/16. The internal logic of *detection module* only differs. The first configuration (*config1*, as decribed in section 4.2) is implemented for Flush+Reload attack detection. The second configuration (*config2*, as decribed in section 4.3) is based on *config1* and has a larger sliding window to also detect Prime+Probe attack in which two timer instructions are separated from a longer distance due to greater number of memory accesses measured. As expected, *config2* needs more hardware resources due to additional storage and pattern matching logic. Let us note that this implementation result is given for two specific and simple configurations of a reconfigurable hardware. For a real usage, we should prepare a reconfigurable hardware with sufficient resources able to adapt to future needs.

TABLE 1. RESOURCES UTILIZATION AND DETECTION CAPABILITY OF TWO CONFIGURATIONS OF REHAD DETECTION MODULE

	config1	config2
Sliding window size	3	8
Flush+Reload Detection	yes	yes
Prime+Probe Detection	no	yes
Look-Up Tables (LUTs)	208	215
Flip-Flops (FFs)	65	70
BRAMs	0	0

5. Conclusion and Future Work

In this paper, we introduced REHAD, an architecture that uses a reconfigurable *detection module* and a hardware/software co-design for software-based CSCAs detection. We proposed to benefit from the hardware parallelism to make a high-frequency microprocessor work with a low-frequency reconfigurable hardware. We implemented this architecture on FPGA and proved that it can successfully detect two CSCAs, Flush+Reload and Prime+Probe with different configurations on the *reconfigurable detection module*. Futhermore, we believe that REHAD has the capability to evaluate and detect other attacks, such as other microarchitectural timing attacks, transient attacks and return-oriented programming attacks which have very short and repetitive instruction sequences.

For future work, we plan to improve our *detection module* and evaluate the performance and area overhead of REHAD architecture. We also plan to work on the implementation of the *trusted software kernel*, to include new functionalities such as the transfer of runtime information to the *detection module* regarding the execution environment, and the process currently executed in order to perform hardware-software co-detection.

We also want to take into account multi-core systems and context changes in order to detect the use of another counting thread to replace accuracy timer in the CSCAs as well as attacks that use multiple threads. We finally plan to investigate a proper method aiming at including in this architecture existing hardware counters such as HPCs and benefit from state of art HPC-based detection.

References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019, pp. 1–19.

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

[4] Y. Lyu and P. Mishra, "A Survey of Side-Channel Attacks on Caches and Countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, Mar. 2018.

[5] VectorBlox, "Orca," Dec. 2019. [Online]. Available: https://github.com/VectorBlox/orca

[6] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, Aug. 2014, pp. 719–732.

[7] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology CT-RSA 2006*, vol. 3860. Berlin, Heidelberg: Springer, Feb. 2006, pp. 1–20.

[8] Y. Yarom, "Mastik: A Micro-Architectural Side-Channel Toolkit," 2016. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/

[9] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Preventing Microarchitectural Attacks Before Distribution," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 377–388.

[10] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding, "SCADET: a side-channel attack detection tool for tracking prime+probe," in *Proceedings of the International Conference on Computer-Aided Design - ICCAD '18*. San Diego, California: ACM Press, 2018, pp. 1–8.

[11] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, Dec. 2016.

[12] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Dec. 2014, pp. 216–228.

[13] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2015, pp. 651–661.

[14] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 26, no. 2, pp. 203–215, 2007.

[15] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[16] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable Hardware for High-Security/ High-Performance Embedded Systems: The SAFES Perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, Feb. 2008.