TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

---

# Using dynamic partial reconfiguration as a security mechanism against cache based side channel attacks

---

*Author:*
Alexandros SKYVALOS

*Thesis Committee:*
Assoc. Prof. Sotiris IOANNIDIS
Prof. Apostolos DOLLAS
Prof. Eftychios KOUTROULIS

*A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer*

*in the*

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

July 2, 2024

TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Using dynamic partial reconfiguration as a security mechanism against cache based side channel attacks**

by Alexandros SKYVALOS

Side Channel Attacks pose a significant threat to modern processors, since they are able to steal information by observing the normal operation of the system. Cache based SCAs are especially dangerous as they can extract vital information by simply monitoring the state of the processor's cache, while simultaneously being used in a plethora of other well-known attacks, such as Spectre and Meltdown. Many defenses have been proposed towards the mitigation of these attacks, however they all come at a cost in either complexity, performance or resource usage. In this thesis we propose an effective solution, that is able to successfully detect and mitigate cache-based SCAs at the hardware level, without introducing performance penalties and any significant area overheads. Our solution leverages the dynamic partial reconfiguration feature of modern FPGAs to introduce a reconfigurable cache. We implement this scheme on an open source RISC-V processor (CVA6), which we modified to support multiple cache configurations that can be swapped during run-time. The cache reconfiguration is handled at the hardware level and does not interrupt the system's normal operations, making it completely transparent to the software components running on top of the processor. We are able to detect impending attacks by monitoring accesses to timing resources, at which point we switch the cache configuration. We are able to show that by reconfiguring the cache targeted by these attacks we can successfully prevent them from extracting information. Our solution doesn't impact the processor's performance and requires minimal additional resources to implement, making it a viable defense against these types of attacks.

TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Using dynamic partial reconfiguration as a security mechanism against cache based side channel attacks**

by Alexandros SKYVALOS

Οι επιθέσεις πλευρικών καναλιών αποτελούν σημαντική απειλή στους σύγχρονους υπολογιστές, καθώς είναι σε θέση να υποκλέψουν πληροφορίες παρατηρώντας την κανονική λειτουργία του συστήματος. Οι επιθέσεις που στοχεύουν την κρυφή μνήμη είναι ιδιαίτερα επικίνδυνες καθώς μπορούν να εξαγάγουν ζωτικές πληροφορίες παρακολουθώντας την κατάσταση της κρυφής μνήμης του επεξεργαστή, ενώ ταυτόχρονα χρησιμοποιούνται σε μια πληθώρα άλλων γνωστών επιθέσεων, όπως το Spectre και το Meltdown. Έχουν προταθεί πολλές άμυνες για τον μετριασμό αυτών των επιθέσεων, ωστόσο όλες εισάγουν κόστος είτε σε πολυπλοκότητα, απόδοση ή χρήση πόρων. Σε αυτή τη διπλωματική εργασία προτείνουμε μια αποτελεσματική λύση, η οποία είναι σε θέση να ανιχνεύει και να μετριάζει τέτοιου είδους επιθέσεις, χωρίς να επιβαρυνει την απόδοση του συστήματος. Η λύση μας εκμεταλλεύεται τη δυνατότητα δυναμικής μερικής αναδιαμόρφωσης των σύγχρονων FPGA για να εισάγει μια επαναδιαμορφώσιμη κρυφή μνήμη. Το εφαρμόζουμε σε έναν επεξεργαστή ανοιχτού κώδικα RISC-V (CVA6), τον οποίο τροποποιήσαμε για να υποστηρίζει πολλαπλές διαμορφώσεις κρυφής μνήμης που μπορούν να εναλλάσσονται κατά το χρόνο εκτέλεσης. Η αναδιαμόρφωση της κρυφής μνήμης γίνεται σε επίπεδο υλικού και δεν διακόπτει την ομαλή λειτουργία του συστήματος, καθιστώντας το διαφανές προς το τρέχων λογισμικο. Είμαστε σε θέση να ανιχνεύσουμε επικείμενες επιθέσεις παρακολουθώντας τις προσβάσεις σε πόρους χρονισμού, οπότε αλλάζουμε τη διαμόρφωση της κρυφής μνήμης. Αποδεικνύουμε ότι με την αναδιαμόρφωση της κρυφης μνήμης που στοχεύουν αυτές οι επιθέσεις μπορούμε να αποτρέψουμε την εξαγωγή πληροφοριών. Η λύση μας δεν επηρεάζει την απόδοση του επεξεργαστή και απαιτεί ελάχιστους πρόσθετους πόρους για να υλοποιηθεί, καθιστώντας την μια βιώσιμη άμυνα έναντι αυτών των τύπων επιθέσεων.

# *Acknowledgements*

First and foremost I would like thank Andreas Brokalakis for his guidance and patience throughout this entire process. His support was invaluable and without him this entire work wouldn't be possible.

I would also like to thank Prof. Sotirios Ioannidis for giving me the opportunity to work on this subject while also Prof. Apostolos Dollas and Prof. Eftychios Koutroulis for being part of my thesis committee.

Lastly, I would like to express my gratitude to Dr. Georgios Christou and Dr. Konstantinos Georgopoulos for their advice, and the rest of the MHL lab for providing the necessary tools and technical support during the work for this thesis.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| **SCA** | **S**ide **C**hannel **A**ttacks |
| **ALU** | **A**rithmetic **L**ogic **U**nit |
| **ASIC** | **A**pplication **S**pecific **I**ntegrated **C**ircuit |
| **BRAM** | **B**lock **R**andom **A**ccess **M**emory |
| **CPU** | **C**entral **P**rocessor **U**nit |
| **CS** | **C**omputer **S**cience |
| **DDR3** | **D**ouble **D**ata **R**ate type **3** memory |
| **DRAM** | **D**ynamic **R**andom **A**ccess **M**emory |
| **CSR** | **C**ontrol **S**tatus **R**egisterr |
| **FF** | **F**lip **F**lops |
| **FPGA** | **F**ield **P**rogrammable **G**ate **A**rray |
| **DFX** | **D**ynamic **F**unction e**X**change |
| **ICAP** | **I**nternal **C**onfiguration **A**ccess **P**ort |
| **RM** | **R**econfigurable **M**odule |
| **RP** | **R**econfigurable **P**artition |
| **PTW** | **P**age **T**able **W**alker |
| **HDL** | **H**ardware **D**escription **L**anguage |
| **HLS** | **H**igh **L**evel **S**ynthesis |
| **UART** | **U**niversal **A**synchronous **R**eceiver **T**ransmitter |
| **GPIO** | **G**eneral **P**urpose **I**nput **O**utput |
| **SPI** | **S**erial **P**eripheral **I**nterface |
| **LUT** | **L**ook **U**p **T**able |
| **SoC** | **S**ystem **o**n **C**hip |
| **PL** | **P**rogrammable **L**ogic |
| **PS** | **P**rocessing **S**ystem |
| **RAM** | **R**andom **A**ccess **M**emory |
| **ROM** | **R**ead **O**nly **M**emory |
| **SDK** | **S**oftware **D**evelopment **K**it |
| **SSD** | **S**olid **S**tate **D**rive |
| **FSM** | **F**inite **S**tate **M**achine |

*Dedicated to my family and friends…*

# Chapter 1

# Introduction

## 1.1 Motivation

Computing systems are behind every aspect of modern life. They process and store massive amounts of all sorts of data, including information related to the safe operation of critical systems or important personal accounts such as medical records. This makes any computing system a potential target for malicious actors and an immense amount of attacks manifests every year.

One of the most serious security threats are side channel channels attacks. These attacks do not directly target a vulnerability of a computing system but gather data by observing and analysing its normal operation properties, such as power consumption and voltage leakage of its electronic components or timing information related to the execution of computing tasks. The latter are particularly serious as they can be launched successfully without access to the physical hardware nor with exceptional privileges on the software domain simply by observing the timing of events on some particular units of modern processors.

Almost all of these attacks, at some point of their execution, try to hijack information from the cache subsystem. To do so, they use commonly available knowledge about the structure of the cache subsystem (e.g. overall cache size and organization) to construct a process that measures the cache access time. This enables them to deduce which specific parts of the cache have been accessed by a victim process. The popularity of such attacks has surged in recent times [1], especially since the now famous Spectre attack [2] has been published.

## 1.2 Thesis and Contributions

Although different approaches have been proposed as mitigation measures against cache-based side channel attacks, our main thesis is that as long as the cache subsystem remains static, an attacker will eventually find a way to succeed. As such, we propose a dynamic cache organization that changes either in time or as a result of a detected event. This will render any such attack useless since it is constructed according to the specifications of a cache system in order to be successful. Rendering the cache subsystem organization dynamic requires changes to the actual hardware (any software approach will either have detrimental effects on the system performance or add additional layers to the attack surface) and we propose the use of dynamic partial reconfiguration to achieve this.

Indeed, in this work, we employ an open source RISC-V processor design (CVA6 [3]) and demonstrate on actual hardware that cache-based side channel attacks can be successfully launched on its L1 data cache. We modify this cache and implement a number of caches with different microarchitectural characteristics (such as different sizes and organization) as dynamic reconfigurable modules that can be loaded to the processor without disrupting its operation. The reconfiguration process happens at the hardware level and the software (Linux OS and applications) that is executed on the processor is completely unaware of this activity. By analysing the characteristics of the attacks, we identified that in order to be successful they need access to very fine timing information, which can be supplied only through special purpose registers of the architecture measuring processor cycles. We implement at the hardware level a mechanism to detect accesses to this processor resource and use this data as an indicator of a launched attack.

Overall, our system is able to successfully mitigate all cache-based side channel attacks. Our detection mechanism does not produce false negative detections, however it may provide false positives ones. Since the operation of the system is not disrupted by a cache reconfiguration, this does not cause any issues. Furthermore, we investigated the performance cost of this process while the processor was running typical benchmarks and found it to be insignificant. As an additional layer of protection against attacks that have not yet been reported, we added a timing mechanism for the reconfiguration process that enables the system to periodically modify its cache organization, therefore attacks that try to gather system information in order to launch an

attack will not easily succeed. We analysed the performance cost of this process as well and provide a cost/benefit analysis (balancing performance cost against increased security).

Lastly, it should be noted that our proposed solution is purely based in hardware and does not require any changes on the firmware, operating system or application layer. On one hand this provides significant performance benefits compared to solutions that are software-assisted. On the other, software components do not need to be modified (therefore it is a solution that can be immediately applied) and it does not introduce any additional vulnerabilities or increase the attack surface.

## 1.3 Thesis Outline

- **Chapter 2 - Cache based side channel attacks and mitigation strategies:** This chapter provides some background information on side channel attacks, more specifically cache based side channel attacks, some of their uses and the most common mitigation techniques against them.
- **Chapter 3 - Attack Design and Proposed Mitigation:** This chapter provides details behind the design of two cache based side channel attacks, as well as our proposed countermeasure against them.
- **Chapter 4 - Implementation:** This chapter provides details on the architecture of our design and our design choices. More specifically, it details the CPU core we used, the modifications we added to the core, and the implementation of two cache based side channel attacks.
- **Chapter 5 - Results:** Chapter 5 contains the results of the benchmarks we ran on our implemented design, showing the affected performance, power and timing information. Additionally it provides an evaluation of the effectiveness and cost of our solution compared to other related work.
- **Chapter 6 - Conclusions and Related Work:** Conclusion and future ideas that can expand on this work.

# Chapter 2

# Cache based side channel attacks and mitigation strategies

In this section we provide a comprehensive background on side channel attacks (SCA) in order to understand the significant risk they pose to modern computer systems. We begin with the history of SCA, followed by the most well-known research that has been published on them, and their most common implementations. We then analyze the inner-workings of cache based SCA, their most common variations, and provide mitigation strategies that have been proposed in the relevant literature.

## 2.1 Side channel attacks

Historically, the first reported side channel attack was conducted in 1965 when British Intelligence Officials of MI5 attempted to break a cryptographic cipher used by the Egyptian Embassy [4]. By placing a microphone near a roto-cipher machine they were able to listen to the click-sounds the machine produced and successfully deduce the position of the machine rotors.

The first research on SCA was introduced in 1996 by P. Koch [5] who showed the threat they could pose to cryptographic systems. Malicious attackers could deduce specific keys used in cryptographic algorithms such as RSA [6] and Diffie-Helman [7], by measuring the amount of time required to perform certain key operations. Subsequently, in 1999 another paper by P.Koch [8] demonstrated the ability to gather information on the secret keys by monitoring the power consumption of the system during cryptographic operations. Daniel J. Bernstein (2005) [9] and Osvik et al. [10] provided the first implementations of cache based side channel attacks on the AES encryption

algorithm, by capitalizing on the timing variations in cache memory operations.

In the following years more extensive research has been published on the different techniques side channel attacks can use to gather information. Zhou and Feng [11] provide an extensive breakdown of different side channel attack implementations that emerged in the years following P. Koch's publication. Ge et al. present [12] a detailed breakdown of microarchitectural timing attacks and their countermeasures. Below we provide a list of the most common variations of side channel attacks:

**Timing attacks** : These attacks are based on the amount of time certain computations may take to complete. They can be used to measure the execution time of cryptographic algorithms or database queries. By measuring the variations in execution time, the attacker can infer certain information about the data being processed. One of the most common cases, where this technique is applied, is in cache-based side channel attacks. They capitalize on the variations in execution time of memory operations, as a result of cache misses and hits. A malicious user can deduce information about the victim's cache state by measuring the time it takes to access certain data saved in memory. If the access time is low the user can conclude that it is located inside the cache, while if the access time is high it would be located inside the main memory.

**Power analysis attacks** : In these types of attacks the attacker monitors the power consumption of the system's hardware during certain cryptographic operations and therefore gather useful information about the secret keys being used.

**Electromagnetic analysis attacks** : Similarly to power analysis, these attacks monitor the electromagnetic radiation emitted by a device during its operation, to gather information on certain data being used.

**Acoustic cryptanalysis** : These attacks work by analyzing sounds being emitted by a device during certain cryptographic operations. They can use audio monitoring devices to monitor sounds such as keyboard keys being used or other internal computer components that produce sounds during a system's operations. By breaking down these sounds, the attacker can gather significant information on the secret keys being used during these operations.

**Fault attacks** : In this type of attack, the malicious user induces certain faults in a system in order to cause some unwanted behavior which the attacker can then capitalize on to gather sensitive information.

All these types of attacks don't require any direct access to the victim's code and data nor do they depend on any actual design error or vulnerability. They simply rely on just observing different characteristics of a system, which makes them especially dangerous since they can't be that easily detected. In our work, the main type of side channel attack we will be focused on are cache based side channel attacks, which have become a serious threat in recent times with the emergence of attacks like Spectre [2] and Meltdown [13].

## 2.2 Cache based side channel attacks

In any computing system that executes more than one software process, hardware resources are typically shared. This becomes even more pronounced in multi-user environments and to an even greater scale in modern era, in cloud deployments. CPU caches are one of the components that are shared among different processes and this poses a serious security issue, since caches are susceptible to cache based side channel attacks, as mentioned in the previous section. In this section, we are going to provide details on how these attacks can be realised.

In order to explain how cache based SCA work, we must first provide a quick overview of how CPU caches work. A CPU Cache is a small memory component that serves as a buffer between the main memory and the CPU. It stores the most frequently used data and is much faster than the main memory. Most caches are composed of multiple hierarchy levels (L1, L2, L3...), where each subsequent level is slower and bigger in size. Each level is composed of cache lines which are memory blocks of fixed-size that store data from the main memory. The way in which data from the main memory is mapped onto specific cache lines is called cache associativity. For example a 4-way associative cache allows the storage of a memory block onto 4 possible cache lines, which comprise of a cache set. A direct associative cache allows the storage of a memory block only in one specific cache line. When the CPU requests data it first looks into the cache, starting from the first level and going down the hierarchy. If the data is located in the cache we have a cache hit and the cache line is fetched. If the data isn't located in the cache we have a cache miss, and the data is fetched from the main memory with a significant time penalty.

This difference in timing between a cache hit and cache miss is they key to

cache based SCA. By measuring how long a memory operation takes, a malicious user can figure out whether certain data is located inside the cache or not. Certain cryptographic algorithms for example, may access specific lines in the cache depending on the bits of a key. With the knowledge of the way those algorithms are implemented, an attacker can find out which cache lines were used by the algorithm and therefore deduce the bits of the key. In general, any software process that modifies the cache in a certain way, can be taken advantage of by a malicious attacker who can observe the cache's state and therefore infer some sensitive information about the user. Below we present a list of the most common variations of cache based SCA:

**Prime and Probe** : The attacker first primes the cache by filling specific cache sets with his own arbitrary data. The victim then proceeds to run its own program, which may access specific cache lines in the cache and therefore evict some of the attacker's primed cache sets. The attacker then probes the cache again by accessing the previously loaded cache sets and finds out which ones the victim used, by observing if they take longer to load than before.

**Flush and Reload** : This type of attack requires some type of shared resource between the attacker and victim (e.g. shared memory pages). The attacker can flush specific cache lines from the cache by using a command like *clflush*. The victim then runs his own program, accessing specific cache lines in the process. The attacker can then *reload* the cache by accessing specific cache lines and find out which cache lines have been used by the victim, due to their short access time.

**Evict and Reload** : This attack is similar to flush and reload, but instead of flushing specific cache lines with an in-built command, the attacker evicts specific cache lines by filling them with his own data.

**Evict and Time** : In this type of attack, the attacker first lets the victim program run normally and measures the time taken to complete. The attacker then evicts some cache lines and lets the victim run again. By measuring the time taken by the victim to run the program again with the evicted cache lines, the attacker can deduce whether those cache lines were used by the victim or not.

**Flush and Flush** : Similarly to Evict and Time, the attacker lets the victim program run and measures its execution time. He then proceeds to flush specific cache lines and lets the victim run again. By measuring the difference

in execution times, the attacker can find out if those cache lines were used by the victim or not.

There have been numerous research papers on the different implementations of cache based SCA. Yarom and Falkner [14] show the usage of the flush and reload attack on the Last Level Cache of an x86 intel processor (LLC) in order to extract the private encryption keys of a victim running the GnuPG 1.4.13 implementation of RSA. Liu et al. [15] show the usage of a Prime and Probe attack on the LLC of an x86 processor by extracting the private keys of different GnuPG implementations. Ristenpart et al.[16] show the implementation of side channel attacks across virtual machines that share the same hardware resources, while Irazoqui et al. [17] managed to complete a cache based SCA that works between different processors. Su and Zeng [1] and Musthaq et al. [18] provide an extensive breakdown of cache based side channel attacks.

In 2018, two new types of attacks, Meltdown and Spectre [19] were discovered, which received widespread attention after showing a key vulnerability in the architecture design of almost all modern CPUs. Spectre attacks [2] work by exploiting the speculative execution process of modern processors, which is a feature that allows processors to execute instructions ahead of time, predicting the outcome of certain branch instructions in order to increase performance. By training the branch prediction mechanism an attacker can force the processor to execute certain instructions speculatively, which then access privileged data. That data can then be leaked through certain side channels with the help of cache based SCA like Flush and Reload.

Similarly, Meltdown attacks [13] exploit the out-of-order execution feature of certain processors, which allowed processors to execute instructions ahead of time for increased performance. By allowing the processor to speculatively execute certain instructions an attacker can gain access to the kernel memory of a system, completely bypassing the memory isolation between user and operating system. The attacker can then leak the results of those instructions through side channels with the help of cache based SCA.

It is important to notice, that Spectre and Meltdown attacks target different shared components of the a modern CPU. However, to complete the attack and steal the sensitive data, they rely on the use of a cache based SCA. This is common pattern in side channel attacks and it is one of the reasons why providing methods to mitigate cache based SCAs is so important.

## 2.3 Mitigation strategies

Due to the severe risk cache based SCA present to security, it is essential to find ways to defend against such attacks. Below we provide a list of the most common mitigation techniques.

1. **Information independency** : Cache based SCA require a correlation between the cache state and the secret data being processed by the user. By having the cache state be completely independent of any information processed by the victim, the attacker can't gain any insight onto the cache state and therefore is unable to gather any sensitive information. This can be accomplished for example by certain constant-time techniques in which the execution time of a cryptographic algorithm is constant regardless of the bits of the secret key that is being used. Andrysco et al. [20] implemented libfixedtimefixpoint, a new fixed-point, constant-time math library while Bernstein et al. [21] created a new cryptographic library called NaCl that tries to avoid any data flow between the secret key and load addresses or branch predictions. This method comes at high degree of difficulty to implement and may not work on different hardware platforms.

2. **Time blinding** : Another countermeasure would be to modify the time (cycles) read by the attacker. By providing a virtual clock to the attacker which hides the real access time, it would be impossible for the attacker to time the cache accesses correctly, and therefore unable to gain any insight onto the cache state. Peng Li et al. [22] show this by using Stopwatch which runs three replicas of VMs and imposes the time used for an event by each replica as the median of those 3. Determinator [23] is an OS that provides a deterministic execution framework, where the inner execution times are protected from outside user access. These methods come at a significant downside in performance though, with Stopwatch showing an increase of 280% in network-intensive benchmarks and an increase of 230% in compute-intensive benchmarks. Another method to keep the attacker in the dark about timing information is the use of a "black box" [24] which would hide all internal timing information, while only presenting a single timing result of the entire process.

3. **Time sharing** : Since the attacks require concurrent accesses of the cache between attacker and victim, another solution would be to either provide time-sliced exclusive access to the cache or to carefully manage the transition between time-slices. Godfrey et al. [25] proposed the idea of flushing the

cache during VM switches in cloud computing which comes at a cost of performance, while Varadarajan et al. [26] enforced a minimum time slice for an exploitable component to prevent the attacker from having enough time to inspect the component during a sensitive operation.

4. **Resource isolation** : Another mitigation technique is to limit the shared hardware resources available to the attacker with methods like disabling hardware threading to limit the threads that have access to the cache, or by disabling page sharing to prevent attacks like flush and reload which take advantage of shared memory pages. Another option is to limit the access the attacker has to the cache by partitioning the cache with methods like cache colouring [27], which divides the cache into colored blocks, where specific memory pages are mapped onto certain colored blocks. Godfrey et al.[28] used this method to restrict each VM to a specific cache partition, protecting them from accesses by other VM's, while Kim et al. [29] implemented a limited form of cache coloring with STEALTHMEM by providing stealth pages for storing security-sensitive data. Wang and Lee [30] showed a partition-locked cache (PLcache) which locks sensitive data in specific cache lines, which can't be accessed or evicted by other partitions, preventing any external cache interference. A disadvantage of this type of solution is the performance cost of altering the system's components.

5. **Channel interference** : Most attacks require precise timing information in order to work. Adding noise to the channel would complicate the attacker's attempts at gathering sufficient information. Hu [31] added noise to all clocks as a countermeasure to timing channels, while Zhang et al. [32] set up a bystander virtual machine to inject noise into the L2 cache of the entire virtual machine platform. Vattikonda et al. [33] modified the XEN hypervisor to insert noise into the high-resolution time measurements in VMs by modifying the values returned by the rdtsc instruction. Wang and Lee [30] suggested the random permutation cache (RPCache) which provides a randomised cache indexing scheme and protection attributes in every cache line, while Brickell et al. [34] modified the AES implementation and added noise to the cache operations during the algorithm's execution. Despite the complexity added to side channels by these methods, full protection can't be guaranteed since the attacker can still gather information with enough time and effort. The addition of noise may also degrade the system's performance.

6. **Detection** : Another option is to detect when the attack is about to occur by monitoring certain elements such as performance counters which can provide an indication if someone is about to launch a malicious attack. Mushtaq et al. [35] propose a run-time detection mechanism called NIGHTs-WATCH which uses machine learning modules that gather data from certain hardware performance counters (HPC), that successfully detects Flush+Reload and Flush+Flush attacks on the AES encryption algorithm, with very high detection accuracy (99%). Similar to other mitigation techniques however, this solution adds a significant performance overhead to the system.

# Chapter 3

# Attack Design and Proposed Mitigation

In this chapter we present a detailed breakdown and analysis on the design and implementation of two cache based SCA. We show the necessary requirements and conditions to successfully launch them, and the required knowledge to implement them.

## 3.1 Threat Model

The scenario for our attack is detailed here. We consider a victim application that contains a secret key and is isolated from other processes. We suppose that this application contains either a Trojan or a buggy implementation that causes leakage at the micro architectural level. We assume that the attacker is aware of this leakage and is willing to retrieve this secret key from the victim application. We also assume that the victim and attacker share the same CPU core and that the attacker is aware of the CPU's cache characteristics (set association, cache line size). The attacker and victim applications run on the same thread process and thus time-share the core.

## 3.2 Designing the attacks

In order to successfully launch a SCA, the user needs to have a clear and concise view and significant knowledge on the system's architecture, on which they are planning to execute an attack. For example, in the case of a cache based SCA, the attacker must be aware of the cache's line size, number of sets and set associativity as well as its eviction and filling policies (write through or write back).

In the framework of this thesis we will be working on CVA6 [3], an open source RISC-V processor developed by openHWGroup. It consists of a L1 cache with a size of 32 KBs, comprising of 255 sets with 8-way associativity, meaning each set contains 8 cache lines (or ways) of 128 bits each. The cache has a write-through no write-allocate policy, meaning that data written inside the cache are also stored in the main memory, while data written in main memory are only written to the cache on read misses. The cache's eviction policy is pseudo-random with the use of a Linear Feedback Shift Register (LFSR) whose output is used as the index of the way (line) in the set to be evicted. This type of information is necessary to understand and know how the cache state is modified when launching a SCA.

It should be noted that although the attack as presented here is designed for the specific implementation of the processor at hand, by modifying the key characteristics as documented hereafter, it can be employed to any processor. The only requirement is that the processor's cache microarchitecture is known.

### 3.2.1 Trojan Encoding Scheme

In any type of cache based SCA, there must be an aspect in the victim's system that alters the cache's state in such a way where it is dependent on the value of a secret key. This can happen with certain cryptographic algorithms that may access specific cache lines when the bits of the key are 1, or with a trojan which will evict specific cache lines from the cache according to the value of that key. In our case we take advantage of the latter, in which case we added a trojan to the victim's process, which encodes the secret key inside the data cache.

The way this secret key is encoded is by evicting specific cache lines depending on whether the bit of the key is 0 or 1. For example, a key consisting of the binary representation 01001010 would cause the trojan to evict the 2nd, 5th and 7th cache lines of the data cache. However, due to the fact that the Data Cache in our example employs a pseudo-random eviction policy and the cache lines contain previously stored data, it isn't possible to guarantee the exact cache line to be evicted. For this reason we decided to work at the cache set granularity, meaning the above key would be encoded by evicting all the ways inside the 2nd, 5th and 7th cache sets instead.

Since our data cache in this example consists of 256 sets, we can use each set as the bit of the key, meaning we can encode up to 256 bits of a secret string. Each character of a string consists of 8 bits, therefore we can match each character to an octave of sets. For example, if the secret key is called "Password", the first character "P" translates into 01010000 in ASCII code. This means that the trojan will evict the 2nd and 4th set of the cache, while the rest of the sets remain untouched. Next, the character "a" translates into 01100001, meaning the 10th, 11th, and 16th cache sets will be evicted. This continues until the entire string has been encoded, or until we have reached the maximum number of octaves of sets. The end of the string is simply defined by an octave of no evicted sets.

In order to evict specific cache sets we used an array whose size matched the number of cache lines inside the cache, with each array element having the same size as a cache line. In our example this array consisted of 2048 elements with each element being 16 bytes in size. Whenever we wish to evict a cache set from the cache we read all 8 elements in the array whose index matches that cache set. Because the cache employs a no-write-allocate policy it is necessary to cause a read-miss in every array element to load their data from the main memory into the cache. For example, if we wish to evict the cache set 0, we read the array elements Array[0], Array[256], Array[512], Array[768], Array[1024], Array[1280], Array[1536], Array[1792]. This will cause the data cache to evict the previously stored data from one of the 8 cache lines in set 0, 8 different times, and replace it with the data of those 8 array elements. However, there is no guarantee that all 8 different ways inside the set will be evicted when accessing these elements. Due to the random eviction policy, sometimes the same way will be evicted twice or more, while other ways may not be evicted at all. That's why we had to run this process for many iterations, and define 7 out 8 evicted ways as a sufficient threshold to consider that set evicted. A good visual representation of the correlation between our trojan array and the Data Cache sets is shown in figure 3.1

Now that we have our trojan and its encoding scheme defined, the next step is to find a way to extract the information passed to us by the trojan. To do so we implemented 2 cache based SCA : Prime+Probe and Evict+Reload. The basis for those attacks, as well as the trojan's encoding scheme were taken from Martinoli et al. [36] who successfully implemented a prime+probe attack on the CVA6 processor.

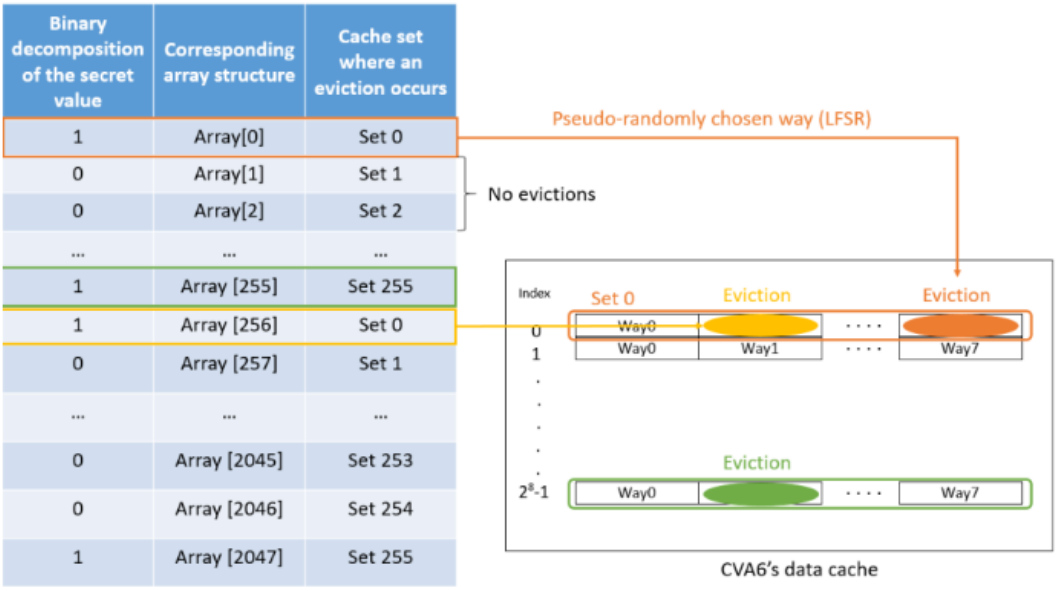


FIGURE 3.1: Trojan array and Data cache correlation taken from [36]

### 3.2.2 Prime+Probe

The prime+probe attack consists of 3 stages :

1. **Prime** : The first stage is to prime the cache by filling its contents with arbitrary data that an attacker would have access to. This is accomplished with the use of an array similar to the one the trojan uses, where each element of the array is 16 bytes and the array consists of 2048 elements. The attacker reads every element of the array which in turn fills every cache line inside the data cache with that array's data.

2. **Victim execution** : After the entire cache has been filled with the attacker's data, the victim process containing the trojan is executed. The trojan will proceed to evict certain cache sets that were previously filled by the attacker in the Prime stage, in accordance to the encoding scheme defined in the previous section.

3. **Probe** : In the final stage, the attacker *probes* the cache after the victim has finished its execution, by reading each element of his array, and measuring the time it takes to access each one, with the help of a timing instruction such as *rdcycle*, which returns the CPU's clock cycles. If the access time is higher than a certain threshold, we have a cache miss and the attacker can assume the cache line corresponding to this element was evicted by the trojan. By measuring all the cache lines that were

evicted by the trojan, the attacker can deduce which cache sets were evicted.

This entire process is repeated for a large amount of iterations to avoid any confusion with other unwanted evictions occurring during the attack, which can be caused by other processes running on the processor at the same time by an OS such as linux. If the same cache set is evicted in at least 75% of the iterations, we can assume that this set was evicted by the trojan and not by any other process. After we were able to define with certainty which cache sets were evicted we can proceed with reconstructing the key, by assigning a bit of 1 to the evicted sets and 0 to non-evicted ones. Afterwards, we decode each octave of sets into a character, until we reach an octave of 0 evicted sets which signifies the end of the string.

A small note here is that during our experimentation with cache set evictions we noticed that cache set number 88 was always evicted 100% of the time. This might have possibly occurred due to a process executing as part of the Linux kernel that was running on our processor. Because the Trojan used octaves of sets to transmit each character of the secret key, the entire octave of sets 88-95 had to be skipped during the trojan's eviction process, and some modifications in the attack's code were required to ignore those sets during the reconstruction of the secret key.

The pseudocode for the prime+probe attack is shown in figure 1

### 3.2.3 Evict+Reload

This attack works similarly to prime+probe with the main difference being that the attacker and victim share a memory component, in this case simulated by a shared array. It consists of 3 stages :

1. **Evict** : The first stage consists of evicting any data from the shared array located inside the cache. This can be made possible with instructions such as *clflush* which can flush data from certain cache lines or even from the entire cache. However, since the processor we used doesn't support such an instruction, we took advantage of eviction by contention, by using another array whose size matched that of the cache. By reading all the elements of this array we are able to evict the previously stored data of the shared array from the cache.

**Algorithm 1** Prime+Probe pseudocode

```
1: function PRIME(Array)
2:     for i ← 0 to sizeof(Array) do
3:         Array[i] ← 16
4:         temp_variable ← Array[i]
5:     end for
6: end function
7: function PROBE(Array)
8:     for i ← 0 to sizeof(Array) do
9:         t1 ← rdcycle
10:        temp_variable ← Array[i]
11:        t2 ← rdcycle
12:        if (t2 − t1 > threshold) then
13:            miss_count[i] + +
14:        end if
15:    end for
16: end function
17: function TROJAN(Secret)
18:    Secret_bits ← ConvertStringToBinary(Secret)
19:    for i ← 0 to sizeof(Trojan_array) do
20:        if (i%256 = 0) then
21:            for j ← 0 to sizeof(Secret_bits) do
22:                if Secret_bits[j] = 1 then
23:                    Trojan_array[i + j] = 150
24:                    temp_variable ← Trojan_array[i + j]
25:                end if
26:            end for
27:        end if
28:    end for
29: end function
30: function VICTIM( )
31:    ...
32:    Trojan(Secret_key)
33:    ...
34: end function
35: function MAIN( )
36:    Prime_array[2048]
37:    for i ← 0 to NR_Iterations do
38:        Prime(Prime_array)
39:        Victim()
40:        Probe(Prime_array)
41:    end for
42: end function
```

2. **Victim Execution** : After the attacker has evicted any data stored from the shared array, the victim executes its own application containing a trojan, which works similar to the one in the prime+probe attack, but instead of evicting cache sets, it accesses them by reading certain elements from the shared array that map to those sets.

3. **Reload** : When the victim has finished execution, the attacker *reloads* the shared array by reading every element, and measures the access time of each with the help of a timing instruction such as *rdcycle*. When the time measured is below a certain threshold we have a cache hit meaning the victim had previously accessed that array element. By measuring all accessed array elements, the attacker can find out which cache sets were used by the trojan.

After this process is run for a certain number of iterations, we reconstruct the secret key in similar fashion to the prime+probe attack, but instead of assigning 1 to evicted sets, we assign 1 to the cache sets that were accessed by the trojan.

The pseudocode for the evict+reload attacks is shown in figure 2

The actual source code for both attacks is written in C and located in `https://github.com/parasecurity/SecurityReconfiguration/tree/main/cva6attacks`. Until the repository is made public, access can be granted upon request.

**Algorithm 2** Evict+Reload pseudocode

```
1: Shared_Array[2048]
2: function EVICT( )
3:     Eviction_Array[2048]
4:     for i ← 0 to sizeof(Eviction_Array) do
5:         Eviction_Array[i] ← 0
6:         temp_variable ← Eviction_Array[i]
7:     end for
8: end function
9: function RELOAD(Array)
10:    for i ← 0 to sizeof(Array) do
11:        t1 ← rdcycle
12:        temp_variable ← Array[i]
13:        t2 ← rdcycle
14:        if (t2 − t1 < threshold) then
15:            hit_count[i] + +
16:        end if
17:    end for
18: end function
19: function TROJAN(Secret)
20:    Secret_bits ← ConvertStringToBinary(Secret)
21:    for i ← 0 to sizeof(Shared_array) do
22:        if (i%256 = 0) then
23:            for j ← 0 to sizeof(Secret_bits) do
24:                if Secret_bits[j] = 1 then
25:                    Shared_array[i + j] = 150
26:                    temp_variable ← Shared_Array[i + j]
27:                end if
28:            end for
29:        end if
30:    end for
31: end function
32: function VICTIM( )
33:    ...
34:    Trojan(Secret_key)
35:    ...
36: end function
37: function MAIN( )
38:    for i ← 0 to NR_Iterations do
39:        Evict()
40:        Victim()
41:        Reload(Shared_Array)
42:    end for
43: end function
```

## 3.3 A Proposed System to Counter Cache-Based SCA Attacks

The design and successful execution of the cache based SCAs as described in the previous sections provide some clear insights. The first one is that although cache-based SCAs can be designed for any cache subsystem, their implementation cannot be generic, but depends on the actual microarchitecture of the cache subsystem of the processor under attack. Knowing (or possibly discovering through properly constructed code) microarchitectural details of the cache such as cache size, cache line size, set associativity and set number are required in order to launch a successful attack.

As such, a malicious user has to set up his attack based on those exact characteristics and perform precise timing measurements to monitor the state of the cache. Our thesis therefore is that if the cache implementation constantly changes, an attacker won't have enough time to gather sufficient information to launch an attack, or will mistakenly launch the attack on stale information. This idea mainly originates from the concept of Moving Target Defense (MTD) [37], which in principal is a cyber-defense technique that revolves around making aspects of a system under protection dynamic in order to thwart attackers that rely on its static nature.

To implement this ever-changing cache there are two main options. The first one requires a cache subsystem that can be configured through software. This means that a "programmable" cache implementation is fixed in hardware and a software component (such as an OS kernel module) properly sets the operation configuration parameters. The second option requires that the actual hardware changes. This is feasible if the overall hardware design is actually reconfigurable or the cache is implemented on a reconfigurable core within a static design.

We focus on the latter option as we are targeting FPGA technology and processors implemented on this technology. Modern FPGAs can be reconfigured during their operation (dynamic reconfiguration) and even more interestingly, there is support for reconfiguring only a portion of the overall FPGA device while the remaining part keeps its normal operation (dynamic partial reconfiguration). Leveraging this capability, we are going to demonstrate that changing the cache implementation can be performed while the rest of the processor remains operational and its computation remains uninterrupted.

Apart from an expected performance advantage of the purely hardware solution, there is another major benefit. The software that is running on top of the processor (both system and application software) is unaware of this reconfiguration process, therefore on one hand absolutely no changes on that layer are required and on the other hand, malicious code cannot affect or compromise the SCA mitigation process.

# Chapter 4

# Implementation

## 4.1 Platform

### 4.1.1 CPU Core

Before starting with the implementation of our proposed dynamic system, we first had to choose a suitable platform for our work. As mentioned in the previous section, we are targeting systems implemented on FPGA hardware and as such, the CVA6 CPU [38], whose cache configuration we detailed in section 3.2, proved an excellent candidate for this work. It is a 6-stage single-issue processor which implements the 64-bit RISC-V instruction set. Specifically, it fully implements the I, M, A and C extensions as specified in Volume I: User-Level ISA V 2.3 as well as the draft privilege extension 1.10. It implements three privilege levels M, S, U to fully support a Unix-like operating system. It contains a Translation Lookaside Buffer (TLB), tightly integrated Data and Instruction caches and a hardware Page Table Walker (PTW). The core's source code is written in SystemVerilog and offers a variety of different Dcache configurations, allowing us to incorporate them in our reconfigurable design. The core's pipeline is shown in figure 4.1.

### 4.1.2 FPGA platform and SoC

The CVA6 processor can be successfully deployed onto the Genesys 2 Development Board [39] by Digilent. The development board is built around the Kintex-7 FPGA from Xilinx and offers a wide variety of peripherals including Ethernet, UART, 1GB of DDR3 memory and a micro SD slot. The CVA6 project [3] provides excellent support for hardware and software components that can be deployed on the board, including a list of system peripherals such

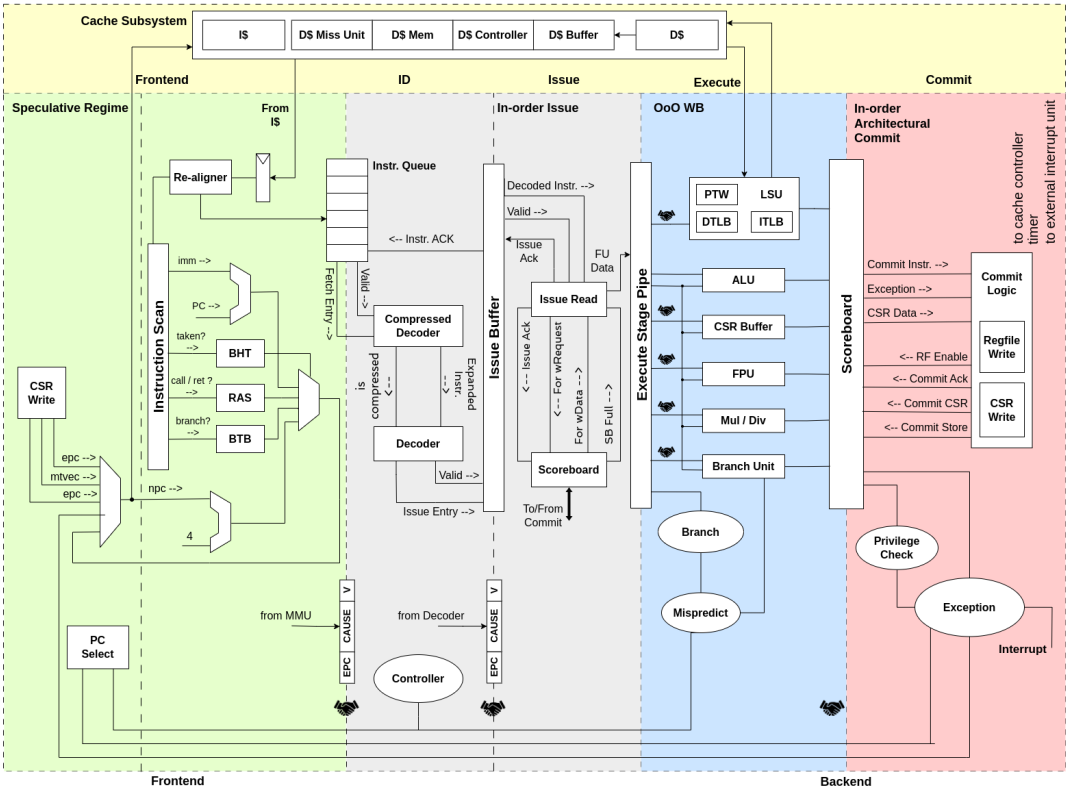


FIGURE 4.1: CVA6 pipeline

as interrupt controllers or debug modules on the hardware side, and the option to run a linux OS on the system. The support of a linux OS enables us to execute common software, benchmarks and our custom applications on the platform and facilitates the use of the system by allowing us to use the board's Ethernet port and thus we can establish SSH connections to use a CLI.

As such, the CVA6 core is part of an entire SoC which additionally includes an Ethernet Controller, UART, GPIO, a Debug Module, a CLINT (Core Local Interrupt controller), PLIC (Platform Level Interrupt Controller), an SPI Interface, a DRAM controller, ROM and a Timer. Each peripheral has an address region assigned to it as shown in image 4.4. The core communicates with the rest of the peripherals through an AXI Crossbar, where it serves as a master alongside the Debug Module, as shown in figure 4.2.

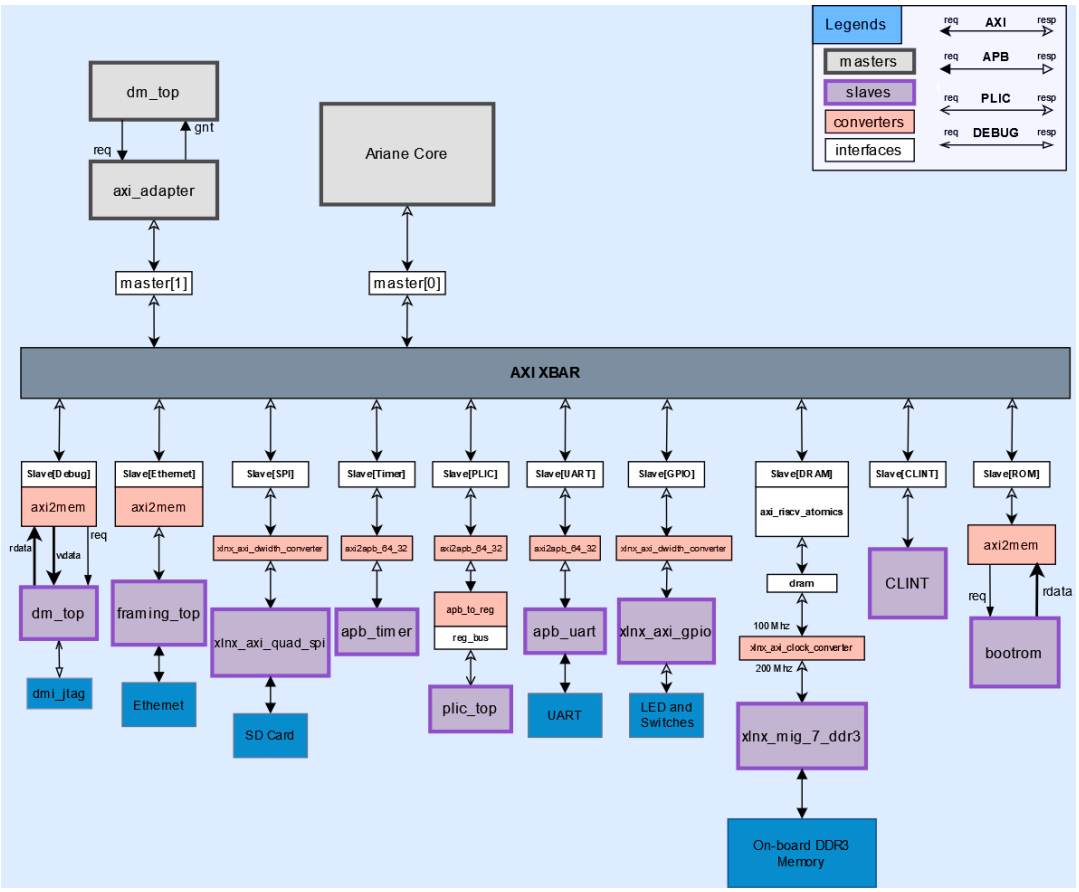


FIGURE 4.2: CVA6 SoC

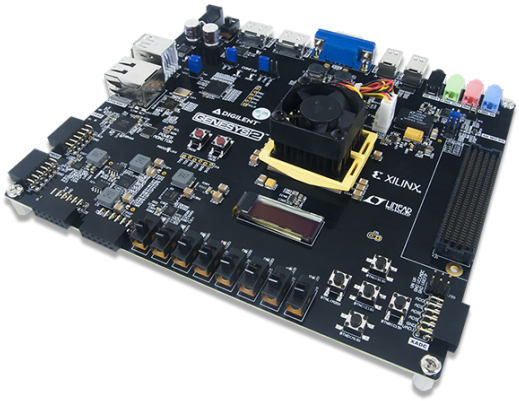

FIGURE 4.3: Genesys 2 Development Board

## Memory Map

| Base | Length | Attributes | Description |
|---|---|---|---|
| 0x0000_0000 | 0x1000 | EX | Debug Module |
| 0x0001_0000 | 0x10000 | EX | ROM |
| 0x0200_0000 | 0xC0000 | | CLINT |
| 0x0C00_0000 | 0x400_0000 | | PLIC |
| 0x1000_0000 | 0x1000 | | UART |
| 0x1800_0000 | 0x1000 | | Timer |
| 0x2000_0000 | 0x80_0000 | | SPI |
| 0x3000_0000 | 0x10000 | | Ethernet |
| 0x4000_0000 | 0x1000 | | GPIO |
| 0x8000_0000 | 0x4000_0000 | EX, NI, C | DRAM |

*(EX: Executable, NI: Non-idempotent, C: Cached)*

FIGURE 4.4: Memory Map of each peripheral on the CVA6 SoC, taken from the CVA6 Docs

## 4.2 Tools used

While working on the deployment of the CVA6 processor on the FPGA platform, we had to employ a few tools which are detailed below.

### 4.2.1 Hardware Tools

#### AMD/Xilinx Vivado

Vivado [40] is a software suite for synthesis and implementation of hardware description language (HDL) designs, developed by AMD/Xilinx. It allows users to design and debug digital circuits running on FPGA devices. It includes an IP integrator tool which enables users to incorporate pre-designed IP blocks into their design. It also features the Dynamic Function eXchange (DFX) tool [41] which offers a user-friendly interface for the partial reconfiguration of hardware designs, which proved very useful and helped with the implementation of our partial reconfigurable design. We used Vivado to be able to modify the core's source code in SystemVerilog and generate bitstreams for the configuration of the FPGA platform. For this work, the Vivado ML (version 2022.2) was used.

### 4.2.2 Software Tools

#### CVA6-SDK

CVA6-SDK [42] is a github repository maintained by OpenHWGroup that holds a list of RISC-V tools as well as all the necessary components for the deployment of a linux OS on the CVA6 SoC [3]. It enables the creation of a custom linux kernel image along with a payload that contains the OpenSBI [43] and the U-Boot bootloader [44]. The kernel image and payload are written to an SD card which can then be plugged into the board in order for the processor's Zero Stage Bootloader (ZSBL) to read the two files and start the linux boot process. By modifying the makefile of this SDK, it was possible to write two partial bitstreams onto the SD Card alongside the payload and linux image. The partial bitstreams could then be read by the ZSBL and loaded into the board's DDR3 memory.

## 4.3 Implementing the attacks

Having set up our platform, we proceeded with the execution of the two SCAs we described in section 3.2. The attacks were compiled with the riscv64-buildroot-linux-gnu-gcc compiler and sent through an SSH connection to a user running inside the OS. Each attack was executed from inside this user's environment in a single thread process and took around 4 seconds to complete when using an iteration count of 200. The Evict+Reload attack could decode the key with a 100% success rate while the Prime+Probe attack had a lower success rate due to the random cache evictions by the OS, which caused the attack to sometimes wrongly decode a few characters of the secret key's string.

We notice that during those attacks a huge amount of calls are made to timing instructions such as *rdcycle* to gather information about the processor's cache state. We can use this to our advantage by monitoring these instructions and using them as a solid indication for when an attack is about to occur. In the following sections we will show how we capitalized on this idea to successfully implement an attack detection mechanism.

## 4.4 Partial reconfiguration

In order to be able to change the characteristics of our processor's data cache we make use of dynamic partial reconfiguration. This is a feature of modern FPGAs allowing them to alter a part of their design while keeping the rest of the device fully operational without interruption. This enables users to modify certain parts of their system during runtime without having to configure the entire device and interrupt its operation.

The user has to first specify a reconfigurable region of logic called a Reconfigurable Partition (RP) with the desirable amount of logic resources. This serves as the area where all the different partial configurations are loaded into. The user can then specify multiple configurations of that region called Reconfigurable Modules (RMs) and generate partial bitstreams for each RM.

Whenever a user wishes to perform a partial reconfiguration, he can load a new partial bitstream through the JTAG port or store it in memory and load it to the device's Internal Configuration Access Port (ICAP) with the help of a controller. The ICAP interface allows users to alter the FPGA fabric

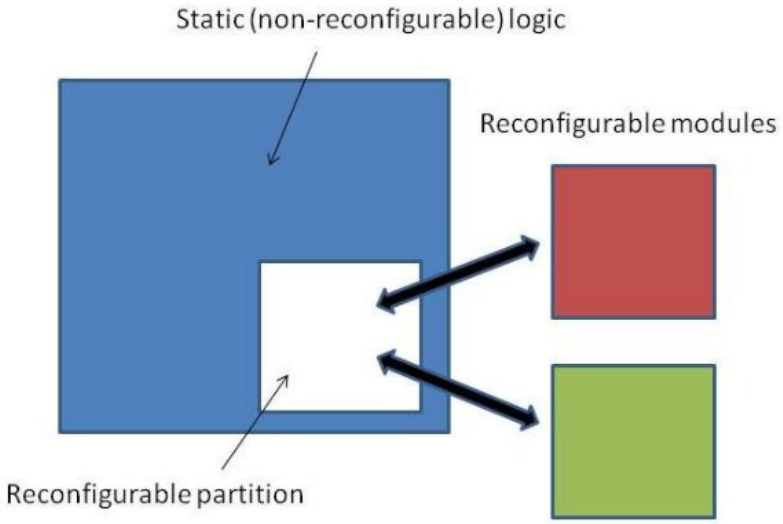


FIGURE 4.5: Partial Reconfiguration visualization taken from [45]

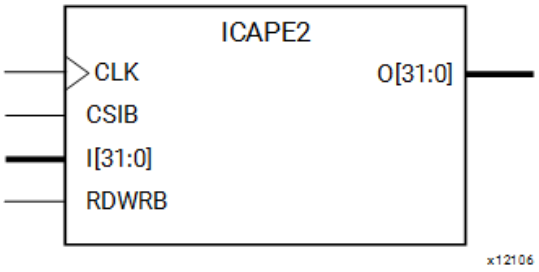


FIGURE 4.6: ICAP interface for 7-series FPGAs

by enabling reading and writing to the registers located inside the FPGA configuration system.

## 4.5 Reconfiguring the Data Cache

### 4.5.1 Creating the Reconfigurable Modules

To start reconfiguring the CPU's cache, we first had to define a region for the different cache implementations. With the help of Vivado's DFX User Interface, the core's Data cache was specified as a Reconfigurable Partition (RP). An area was specified as a partition block (Pblock) shown in figure 4.7 with its properties shown in figure 4.8. This serves as the dynamically reconfigurable region for all the cache configurations. The size of the Pblock was chosen to be large enough to fit many different cache configurations, but could potentially be reduced to limit the device's area overhead. Afterwards,

the DFX Wizard was used to define the desired number of configurations, or otherwise known as reconfigurable modules (RM) for our DCache partition. Two configurations were specified, the default DCache configuration provided by the CVA6 project and a second configuration with half the size (16KBs) and half the set associativity (4-way).

An important note here is that Vivado's DFX is very specific in its requirements for running partial reconfiguration. All reconfigurable modules need to have the same amount of input and output ports, and all ports have to be the same size in every configuration. This limits the available options for different DCache configurations since some of the ports' sizes depend on the cache's properties such as the cache line size or set associativity. In order to comply with those design rules, certain ports which depended on the DCache's set associativity had to be removed, such as the *miss_vld_bits* ports which were used by performance counters to measure cache line invalidations.

### 4.5.2 Storing the partial bitstreams

Continuing forward, the DFX wizard allowed the creation of synthesis and implementation runs for the two DCache configurations, which then generated the partial bitstreams necessary for the reconfiguration of the device. While the initial configuration can be performed manually (e.g. using the JTAG interface), our system has to support automatic partial reconfiguration controlled by a module within the FPGA itself. As a result, determining where to store the partial bitstreams is not a trivial decision and depends on the memory resources of the development board, the specific interfaces supported and the features of the FPGA device. The Genesys 2 board contains two external memories that can potentially be used to store FPGA bitstreams: 1GB of DDR3 memory and 32MB of non-volatile flash memory. Saving the partial bitstreams in flash memory wasn't feasible since reading them would require an AXI Quad SPI interface which employs the STARTUP primitive. However, this is not compatible with loading of partial bitstreams in 7-series Kintex devices [46]. Therefore, the only option available is to use the board's DDR3 memory which is managed by Xilinx's Memory Interface Generator (MIG), part of the CVA6's SoC.

In order to save the partial bitstreams to the DDR3 memory, we first had to save them to an SD Card, which was made possible with the help of OpenHWGroup's CVA6-SDK 4.2.2. We then modified the processor's ZSBL so that

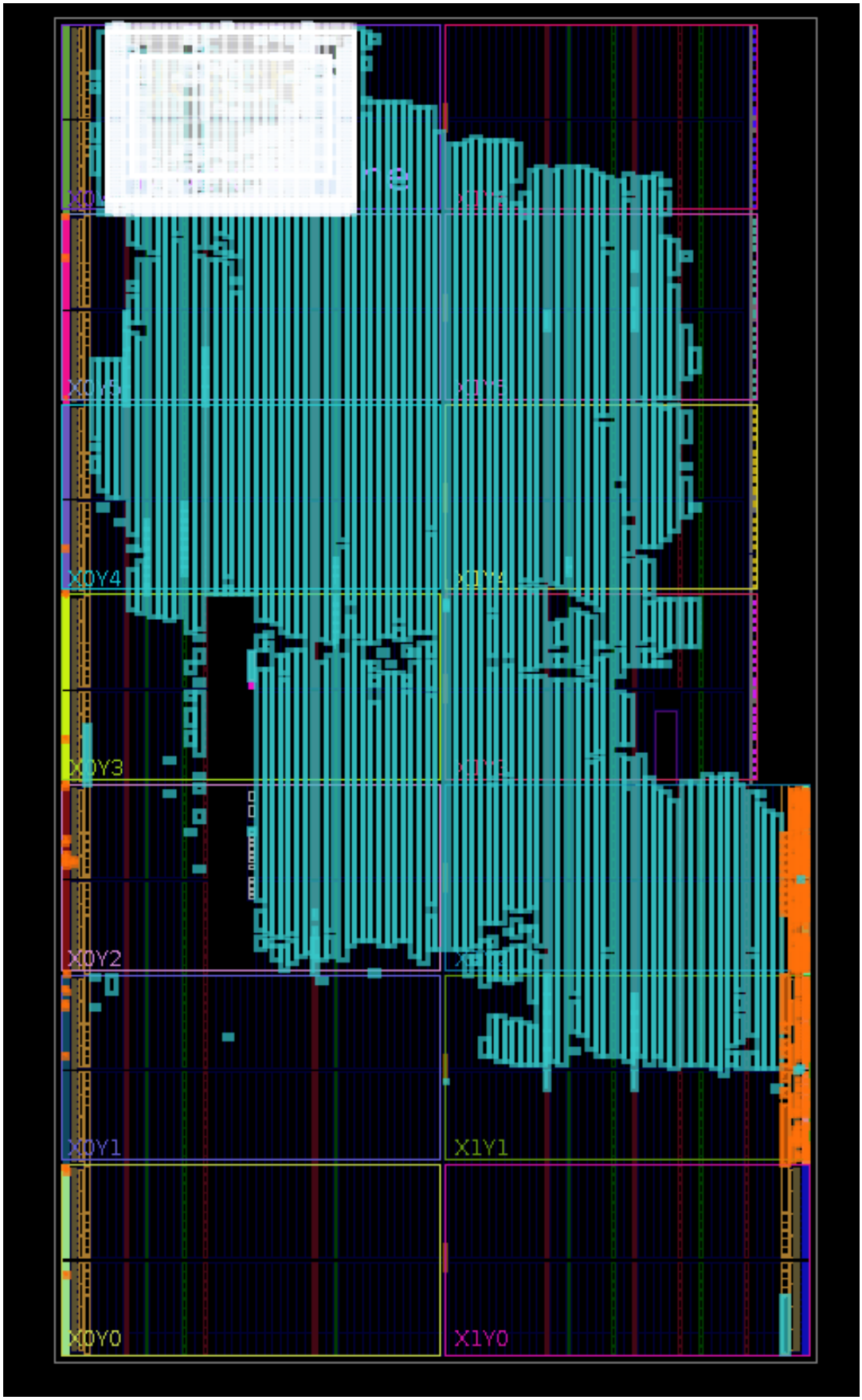


FIGURE 4.7: CVA6 SoC region on the FPGA with the pblock region marked in white

**Physical Resource Estimates**

| Site Type | Parent | Child | Non-Assigned | Used | Available | % Util |
|---|---|---|---|---|---|---|
| Slice LUTs | 5558 | 0 | 0 | 5558 | 10800 | 51.46 |
| LUT as Logic | 5558 | 0 | 0 | 5558 | 10800 | 51.46 |
| LUT as Memory | 0 | 0 | 0 | 0 | 3200 | 0 |
| Slice Registers | 1650 | 0 | 0 | 1650 | 21600 | 7.64 |
| Register as Flip Flop | 1650 | 0 | 0 | 1650 | 21600 | 7.64 |
| Register as Latch | 0 | 0 | 0 | 0 | 21600 | 0 |
| F7 Muxes | 480 | 0 | 0 | 480 | 5400 | 8.89 |
| F8 Muxes | 0 | 0 | 0 | 0 | 2700 | 0 |
| Slice | 1700 | 0 | 0 | 1700 | 2700 | 62.96 |
| SLICEL | 1235 | 0 | 0 | 1235 | 1900 | 65 |
| SLICEM | 465 | 0 | 0 | 465 | 800 | 58.13 |
| Unique Control Sets | 96 | 0 | 0 | 96 | 2700 | 3.56 |
| Block RAM Tile | 24 | 0 | 0 | 24 | 30 | 80 |
| RAMB36/FIFO | 24 | 0 | 0 | 24 | 30 | 80 |
| RAMB18 | 0 | 0 | 0 | 0 | 60 | 0 |
| DSPs | 0 | 0 | 0 | 0 | 60 | 0 |

FIGURE 4.8: Properties of the Pblock region used for the Dcache

it would read these two partial bitstreams from the SD card and write them to predefined addresses of the board's DDR3 memory, specified in the source code of the ZSBL. Choosing the memory addresses proved difficult, since most of the DDR3 memory was reserved for the linux kernel.

As shown in the SoC's memory map 4.4, DRAM occupiess the addresses from 0x80000000 to 0xbfffffff, while the linux kernel occupies addresses 0x80200000 to 0xbfffffff. This leaves only addresses 0x80000000 to 0x801fffff untouched. However since the bitstreams are 1.525.148 bytes each and the address region is only 2MB big, there isn't enough space to save them there.

We eventually arbitrarily chose addresses 0x95000000 and 0x97000000 to save the two partial bitstreams. While experimenting with our reconfiguration attempts, we didn't notice any interference or modifications to these addresses from the linux kernel. However, in the future we can reduce the region the linux kernel occupies, and reserve a memory region at the end of the DRAM address space for the partial bitstreams.

### 4.5.3 Loading new cache configurations

To load the partial bitstreams, the DFX Controller IP [47] was used, which fetches the bitstreams from any type of memory through the AXI4 protocol and loads them into the ICAP of the FPGA Device, with the help of hardware or software triggers. It also provides the necessary signals to decouple or restart the RMs. The controller was added as a third master to the CVA6's AXI crossbar with an AXI data width converter to allow it to communicate with the memory controller of the board's DDR3 memory. In addition to the controller, the DFX Decoupler [48] was connected to the Dcache inside the core, in order to control the values of the output ports of the reconfigurable cache during the reconfiguration process, to avoid any unwanted values reaching the rest of the core during this process.

Whenever we wished to change between a Data cache configuration, a hardware trigger corresponding to a specific RM was enabled and the DFX Controller would read the partial bitstream of that RM from the associated memory address and load it into the FPGA's ICAP. The two partial bitstreams have a size of 1.525.148 bytes each and take around 400.000 clock cycles or 8 ms to finish loading, resulting in a reconfiguration throughput of around 190 MB/s.

An issue that arised during the reconfiguration of the Dcache was the core crashing in situations where the Dcache was still handling outstanding transactions. To solve this problem, certain modifications had to be made to the Load Store Unit (LSU) of the core which sends and receives requests towards and from the Dcache. More specifically the PTW (Page Table Walker), Load and Store Units had their FSMs altered so that when the reconfiguration process was about to occur, those units would stop sending any more requests towards the Dcache until the process had finished. This was made possible with the addition of a signal from the DFX Controller, which would signify when the partial reconfiguration process was about to begin. Additionally some time period had to be given to the Dcache to finish any outstanding requests before starting the reconfiguration process. To accomplish this, a timer was added that would activate an acknowledgement signal towards the Controller when it reached 2000 clock cycles. As soon as the acknowledgement signal was activated the controller would start the partial reconfiguration process. Lastly, a decoupling signal from the Controller was used to enable the Decoupler during the cache's reconfiguration to control any unpredictable values from the Dcache that could harm the rest of the core.

Having successfully implemented the dynamic reconfiguration of the core's Dcache, the next step would be to decide **when** to change between the different configurations in order to successfully thwart an attack. The first thought was to enable partial reconfiguration periodically in a period smaller than the execution time of the entire attack which is around 4 seconds for a total number of 200 iterations. However this wasn't effective at stopping the attack, since each iteration of the attack takes only 1.000.000 cycles or 20ms. Any reconfiguration period that is higher than that duration, would allow for enough iterations of the attack to launch with correct Dcache information, resulting in a successful attack. Enabling partial reconfiguration in a period smaller than each iteration would be detrimental to the core's performance since the entire partial reconfiguration process takes approx. 8ms, meaning that almost half of the core's operation time would be spent in reconfiguration, during which the core can't execute any memory operations.

The second option would be to detect when the attack is about to occur, so that the reconfiguration of the cache can start before the attack is launched, forcing the attacker to execute the attack with the wrong Dcache information. By analyzing the attack's code, we notice that they require precise timing measurements to determine if an element is located inside the cache or not. They accomplish this with the help of special Control Status Registers (CSR) such as *rdcycle* which returns the number of clock cycles executed by the processor's core. We leverage this by adding a counter inside the core at the hardware level which counts the amount of accesses to that specific CSR. If this counter reaches a certain threshold, one of the controller's triggers gets enabled and the partial reconfiguration process begins, which will load a different Dcache configuration to the core, flushing any contents from the Dcache in the process. Because the attacker has to measure the access time of each cache line, and the Dcache consists of 2048 cache lines, we determined 1000 calls to *rdcycle* as a sufficient threshold.

Indeed, with the above implementation we are able to detect 100% of the attacks executed on the core, and to successfully mitigate them. We identify though that there is a potential danger to activate multiple data cache reconfigurations, if the OS itself or other common software relies also in the frequent access of this CSR. We performed a significant number of tests, including the execution of multiple benchmarks that by nature require multiple accesses to timing information and observed that the frequency of accesses to the specific CSR is orders of magnitude lower than what is observed during

a SCA attack. As such, we determined that the threshold set is a safe one to accurately identify a SCA.

It should be noted that by employing this mechanism, we allow for a very low number of false positive detections. This is a side effect of the use of a counter, which will trigger a reconfiguration whenever the threshold is reached, no matter if this has been caused by an attack or by normal system operation. Since, as has been specified, during normal operation the frequency of the CSR accesses is very low, the accumulation of a number of accesses to surpass the threshold requires a very long time period and therefore the reconfiguration mechanism is triggered rarely. The measured impact in the performance of the system is practically zero and as has been described from a functional point of view, there is no disruption or any other issue in the operation of the system.

In addition to the detection mechanism, we added a counter which will enable the core to change between the two Dcache configurations at regular time intervals. We enable this counter 200 seconds after the core has been loaded onto the board to allow the operating system to finish booting before the partial reconfiguration process begins. When the counter reaches the desired time period, a hardware trigger corresponding to a cache configuration different from the one already loaded is activated, which in turn signals the Controller to start loading that configuration from memory. The reasoning behind this periodical switch between configurations is to make it harder for an attacker to gather sufficient information on the cache's characteristics and launch an attack with correct information, in cases where the attack might use other timing measurement methods that aren't detected by our detection mechanism. We experimented with different time periods and analyzed the performance cost for each period, which will be shown in Chapter 5.

## 4.6 Summary of the modified design

To conclude this chapter, we provide a synopsis of all the changes we made to the original CVA6-based design to support our reconfigurable data cache. Our modifications span two levels: the actual CVA6 RISC-V core design and the overall SoC implementation.

At the core level, we modified the data cache subsystem. We added a Decoupler module inside the cache subsystem and connected it to the main

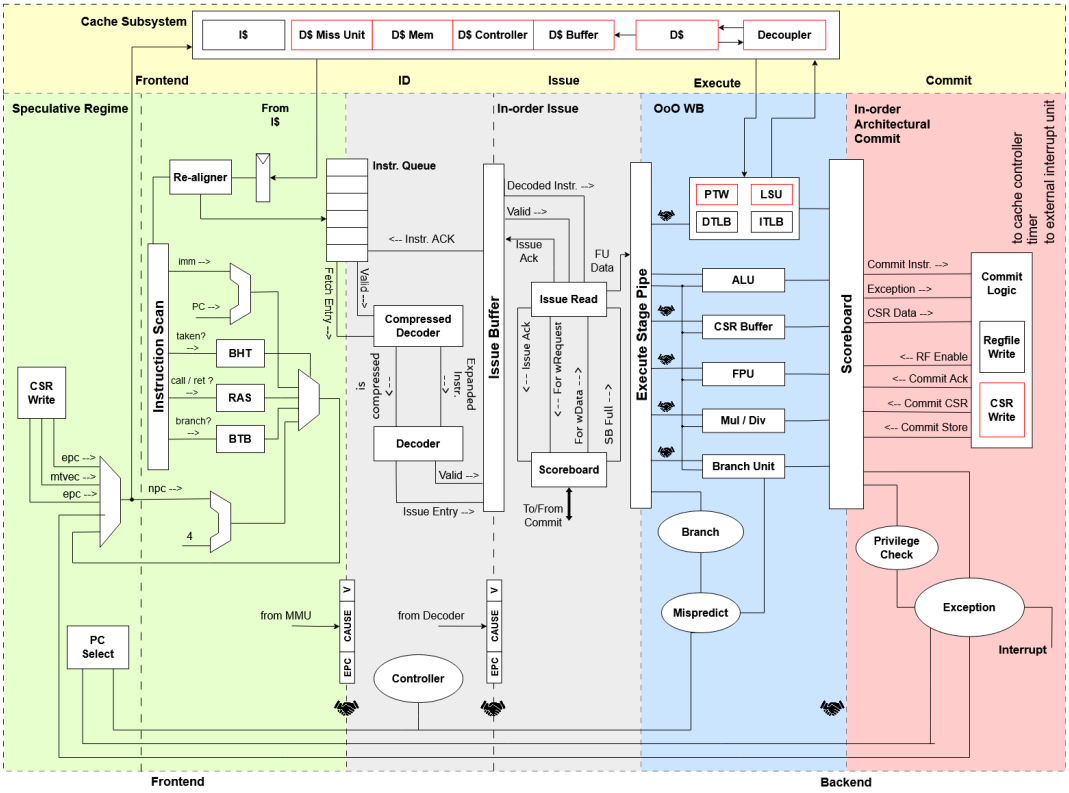


FIGURE 4.9: Modified CVA6 Pipeline

data cache array. Additionally, to properly support the partial reconfiguration process we modified the data cache interface with the rest of the core and produced two different data cache designs (the differences being in their overall cache size and associativity).

To support the required changes imposed by the intricacies of the partial reconfiguration mechanism, we also modified the Load & Store units as well as the Page Table Walker unit, by altering their FSMs. Last but not least, we implemented our attack detection mechanism by adding a counter inside the Control Status Register module which monitors the number of accesses to the cycle register and triggers an alert when they exceed a certain threshold.

All modifications are highlighted in red in figure 4.9.

At the SoC level, we added a DFX Controller as a new peripheral. This peripheral is connected to the AXI crossbar as a master device in order to be able to communicate with the DDR3 memory controller. The new modified SoC is shown in figure 4.10 with the modified modules highlighted in red.

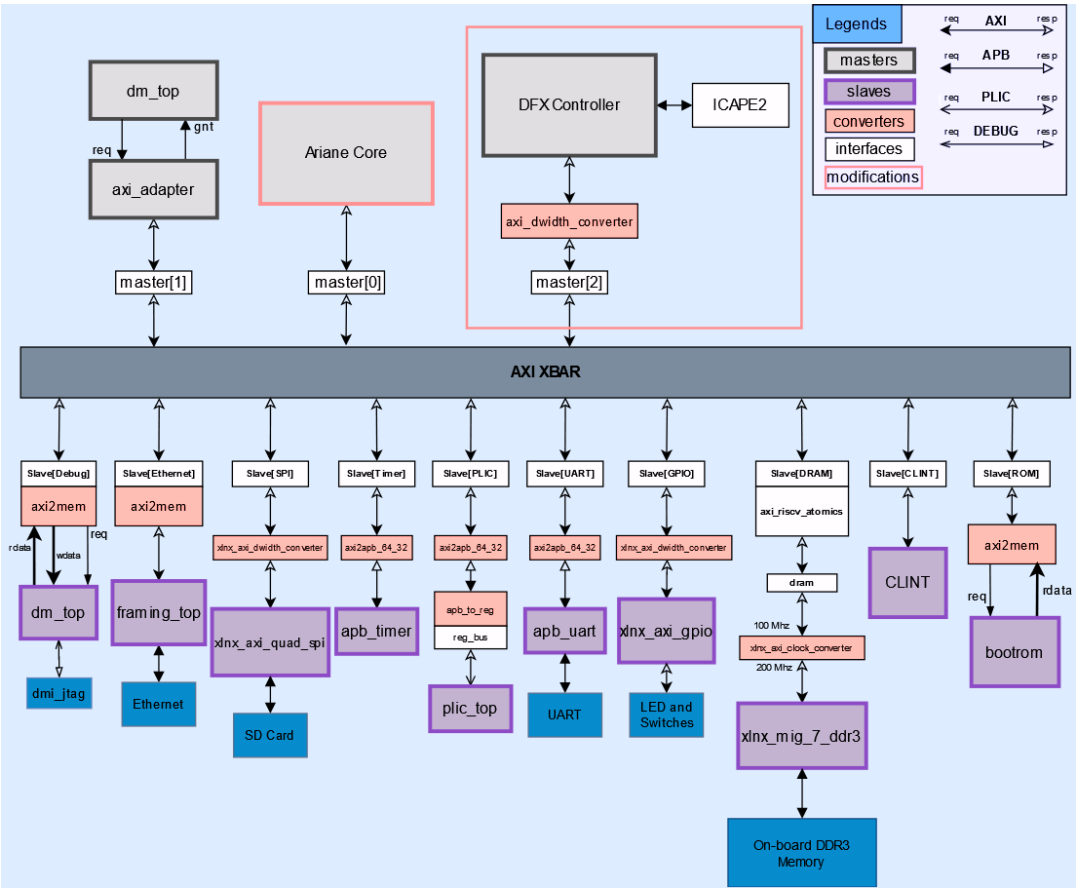


FIGURE 4.10: Modified CVA6 SoC

# Chapter 5

# Results

In this chapter, the results of our work will be presented. More specifically, we are going to analyze the cost of implementing our proposed design in terms of area usage, achieved frequency of operation and power consumption. We will further investigate the impact on the performance of the system by employing commonly available benchmarks. Lastly, we will evaluate these results and compare them to other works related to the mitigation of SCAs.

## 5.1 Area, Timing and Power Consumption Impact

To measure the impact of implementing our proposed design, we compare the initial CVA6 SoC with our modified system in terms of resource usage, timing (clock) and power consumption. To do so, we generated the appropriate reports with the Vivado 2022.2 tool for the original configuration and our modified one.

### 5.1.1 Resource Usage

First, we generated the utilization reports to measure the resources used by the processor system before and after our modifications. The results are shown in table 5.1. It should be noted that these results are derived from the post place and route reports of Vivado and refer to the overall SoC.

We notice a slight increase of 2.5% in the number of LUTs used, 1% in LUTRAM and 2% in FFs, with a slight drop of 1% in IO, while the rest of the resources stayed the same. Our modifications, therefore, have minimal additional resource requirements, making them applicable in practically all scenarios where the CVA6 SoC can be used.

| | Original SoC | Original SoC (%) | Modified SoC | Modified SoC (%) |
|---|---|---|---|---|
| LUT | 70643 | 34.66 | 75637 | 37.11 |
| LUTRAM | 1512 | 2.36 | 1743 | 2.72 |
| FF | 46707 | 11.46 | 49477 | 12.14 |
| BRAM | 50 | 11.24 | 50 | 11.24 |
| DSP | 27 | 3.21 | 27 | 3.21 |
| IO | 117 | 23.40 | 109 | 21.80 |
| MMCM | 3 | 30.00 | 3 | 30.00 |
| PLL | 1 | 10.00 | 1 | 10.00 |

TABLE 5.1: Utilization report before and after our modifications

### 5.1.2 Timing Analysis

We then proceeded to generate the timing report summaries for the processor before and after our modifications which are shown in figures 5.1 and 5.2 respectively. We notice no impact on the Worst Negative Slack (WNS) and Worst Pulse Width Slack (WPWS) which remain the same, while only a small decrease of 0.006ns occurs in Worst Hold Slack (WHS) after our modifications.

According to these results, our modifications do not impact the clock of the CVA6 design and therefore do not incur any performance losses that could manifest as a result of having to decrease the operating frequency of the system. It should be noted that the operating frequency of the CVA6 design is constrained at 50 MHz (as set by the original OpenHW group design for the specific FPGA board).

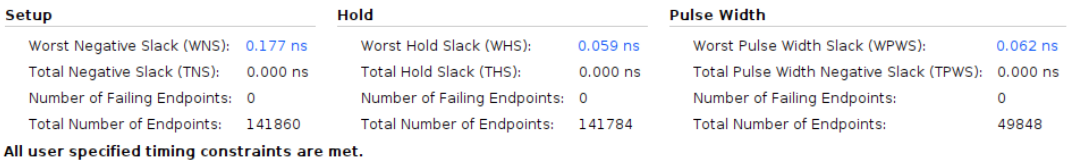


FIGURE 5.1: Original timing summary of the CVA6 SoC

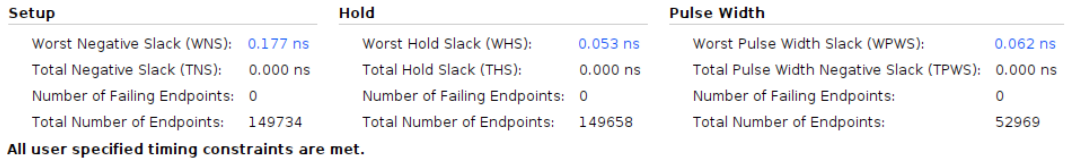


FIGURE 5.2: Timing summary after the modifications to the CVA6 SoC

### 5.1.3 Power Consumption

Lastly, we generated two power estimation reports for the original and modified design with the results shown in table 5.2. We notice a marginal decrease of 0.003 W in the overall power consumption, which can be considered negligible.

| On-Chip | Power (W) | |
|---|---|---|
| | Default SoC | Modified SoC |
| Clocks | 0.162 | 0.165 |
| Slice logic | 0.085 | 0.084 |
| LUT as Logic | 0.079 | 0.078 |
| CARRY4 | 0.002 | 0.002 |
| Register | 0.002 | 0.002 |
| LUT as Distributed RAM | <0.001 | <0.001 |
| F7/F8 Muxes | <0.001 | <0.001 |
| LUT as Shift Register | <0.001 | <0.001 |
| Others | <0.001 | <0.001 |
| BUFG | <0.001 | |
| Signals | 0.111 | 0.108 |
| Block RAM | 0.035 | 0.033 |
| MMCM | 0.322 | 0.322 |
| PLL | 0.133 | 0.133 |
| DSPs | <0.001 | <0.001 |
| I/O | 0.617 | 0.617 |
| PHASER | 0.456 | 0.456 |
| XADC | 0.004 | 0.004 |
| Static Power | 0.176 | 0.176 |
| Total | 2.102 | 2.099 |

TABLE 5.2: Power Consumption before and after our modifications

## 5.2 Performance Impact

In order to measure the performance of the CVA6 core, we used a list of benchmarks provided by RISCV-Tests, a github repository containing a collection of tools designed to test the functionality of RISC-V systems. The benchmarks we used consist of Dhrystone, Towers, VVADD, QSort, Multiply, RSort and Median. These benchmarks were initially designed to run on baremetal systems which meant they were using custom built syscalls and functions. We therefore modified their source code to allow the usage of regular C library calls and compiled them with the riscv64-buildroot-linux-gnu-gcc compiler, the same one used for the attacks.

The compiled benchmarks were sent and executed from a user inside the Linux OS which was deployed on the SoC running on the Genesys 2 Board. To provide the desired comparison, we configured the FPGA with the original CVA6 SoC and ran the benchmarks. We then reconfigured it with our modified design and performed the benchmark runs again.

At this point, it should be noted that when using the detection mechanism, no performance impact is measured. System operation is not disrupted by any means since during normal operation the detection mechanism is not activated. Furthermore, the number of rdcycle accesses when executing benchmarks and during normal operation is very small and therefore even after hours of benchmarking, the threshold for reconfiguration request is not reached.

However, the second protection mechanism that we introduced based on a timer that periodically causes the cache to reconfigure, can indeed incur a performance penalty and this penalty is what we mainly investigate in this section. We use three different reconfiguration periods (namely 0.2s, 0.5s and 1s) as an indicative range. We consider that the longer a system remains unchanged, the less secure it can potentially be. On the contrary, changing the cache configuration too often is expected to cause performance degradation.

We try to quantify this performance impact and for each system (the original CVA6 system and the three systems with the different reconfiguration periods), we measure the amount of cycles required to finish each benchmark. The results are shown in figure 5.3. The baseline for the comparison is considered to be the original CVA6 system and the overhead (in terms of clock cycles) for each benchmark is displayed.

We notice that the performance overhead is directly related (almost inverse linear) with the reconfiguration period. Reducing reconfiguration period by half results in doubling the performance overhead in each benchmark. Typically, for the system that reconfigures its data cache every 1sec, a 1% increase in clock cycles is observed to complete each benchmark, while on the other end, reducing the reconfiguration period to 0.2s results in a performance overhead of approximately 5%. The rsort benchmark is an outlier of this trend, demonstrating higher performance losses spanning from almost 3% to almost 7%.

It should be noted though, that overall the performance overhead is well controlled and it is kept relatively low (below 5% in the worst case). As a result,

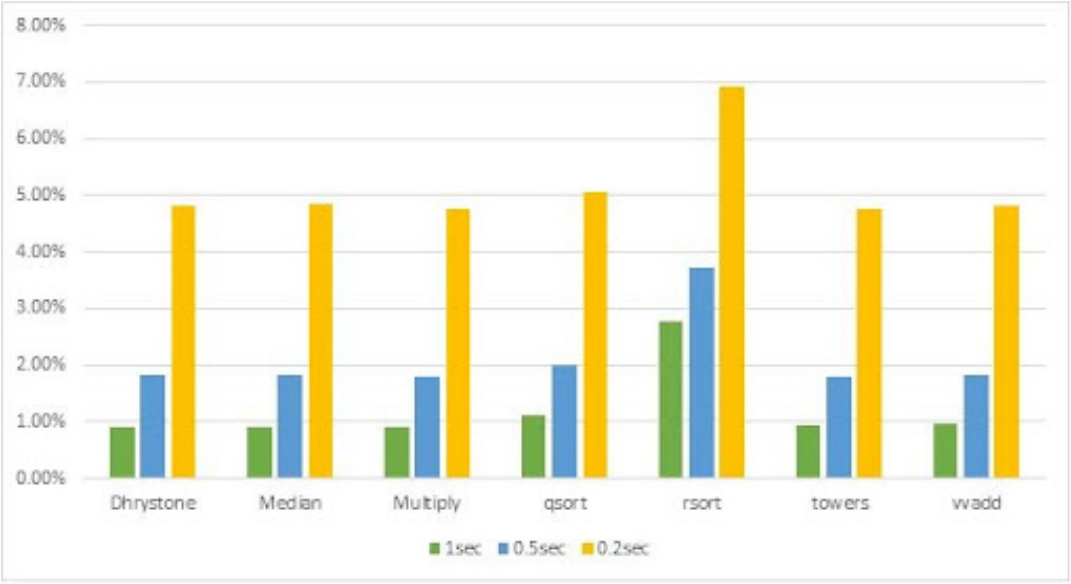


FIGURE 5.3: Cycle overhead for each benchmark for 3 reconfiguration periods

depending on the level of security that a system architect wishes to maintain, this solution can be effective without significant performance compromises.

Two remarks need to be made at this point. First of all, it needs to be stressed that employing the timer mechanism for reconfiguration at this point is entirely optional. Current side channel attacks are 100% successfully detected by our proposed mechanism which does not incur any performance penalties during normal operation (i.e. when the system is not under attack). The periodic reconfiguration mechanism serves only as an additional protection layer for highly critical systems, in order to prevent the design of an attack that may bypass the detection mechanism.

The second remark is related to the observed performance penalty. The benchmark runs demonstrate that the performance overhead is low, ranging from 1% to 5% depending on the reconfiguration period chosen. It should be noted that the CVA6 system is designed with a 50MHz operating clock frequency and a write-through data cache. We would expect that the performance impact will be higher for higher-clocked systems as well as systems that employ more performance-aggressive cache designs or sophisticated prefetching mechanisms. Furthermore, the performance overhead is dependent on the applications themselves and the way they stress the data cache, as for example the rsort benchmark demonstrates. Finally, considering that the use of caches is already problematic with real-time applications, this technique is not suitable for such time-sensitive systems, unless extensive testing and

proper planning has been performed to ensure correct operation.

## 5.3 Evaluation

In order to provide a concise and comprehensive evaluation of our work, we will analyze the performance cost and effectiveness of some of the most significant mitigation techniques against cache-based SCAs that have been proposed in the literature, and compare it to our implemented design. These techniques can range from software-based designs which employ operating systems, compilers or even machine learning, all the way to hardware-based defenses, with drawbacks in either performance, resource usage, or power.

Starting at the software level, a common technique used for the defense against SCAs is cache flushing, which removes any information leakage in the cache which can be potentially exploited by an attacker. Godfrey et al. [28] proposed one such technique, in which they modified the hypervisor in a cloud environment, tainting any potential vulnerable cache state and flushing it in the process. They were able to prevent the execution of SCAs, while incurring an execution overhead of 11% on average due to the constant cache flushing. Similarly, Wistoff et al. [49] implemented this idea on the CVA6 Processor by adding a *fence.t* instruction in the RISC-V ISA, which flushes a series of microarchitectural components that can be part of a side channel. They deploy a custom seL4 microkernel on the processor, which can use this custom instruction to flush the cache and any other components during context switches between different security domains. This method adds a 30% latency in cache operations during every flush, however since the context switch rate is no more than 1Khz, the additional latency is considered negligible.

In addition to their cache flushing defense, Godfrey et al. [28] proposed a cache partitioning scheme in which each VM has limited access to certain cache lines, preventing an attacker from accessing shared cache lines with other VMs and therefore unable to establish a side channel. They implement this through the hypervisor which assigns memory pages to specific partitions during boot time. Each VM is only allowed to access cache lines that belong to their partition. Such a solution however causes a noticeable performance drop, since they limit the amount of resources each VM process has access to. When running the Apache benchmark they show a performance

drop ranging from 2% all the way to 30%, increasing with the number of cache partitions used.

A different technique proposed by Gonzalez-Gomez et al. [50] involves many-core distributed systems which consist of multiple tiles, each containing a number of cores. They propose the implementation of a dynamic task migration mechanism which is able to migrate sensitive applications to more secure cores that don't share any resources with a potential attacker. This is accomplished by modifying the system's Operating System to be able to dynamically assign cores to different threads as well as detect potential attacks by measuring the performance degradation in applications with the help of performance counters, which is a side-effect of some SCAs. In addition, they perform periodical migrations in fixed time periods that are too small for a potential attacker to be able to gather information, in cases where a SCA can't be detected. However, such a mechanism comes at a cost in performance due to the interruption of the execution of an application during the migration step. This causes a performance overhead that ranges from 1.6% on average to a maximum of 9% in the worst case scenario. Another issue that might occur with this solution are false positives that can be caused by benign processes that could cause enough of a performance degradation to trigger a migration.

Some compiler-based solutions have also been presented such as the one by Crane et al. [51]. They use dynamic control-flow diversity to create different implementation methods of executable programs with the help of a compiler, each having a different control-flow with the same execution result. They constantly switch between these different implementations, to prevent an attacker from gathering enough information on the way these programs are implemented. With this method they are able to reduce the accuracy of the keys recovered by the attacker by more than 80% in the best case scenario. However the performance overhead ranges from a factor increase of 1.25x to 2.1x on a secure application all the way to 8x in a benchmark application.

Another technique which has been used is machine learning, in which a neural network can be trained with data from spy processes in order to be able to differentiate between malicious and benign applications. Chiappetta et al. [52] used two versions of machine learning, anomaly detection and neural networks, to successfully detect a spy process that uses the Flush+Reload technique. Additionally, they monitored performance counters and analyzed their data to determine a correlation between a victim and a spy in order

to successfully detect a potential SCA. However, they showed that with a few modifications in the spy's source code, they were able bypass this detection mechanism, with only a small increase in the attack's execution time. The performance overhead caused by monitoring these performance counters was 2.3%.

Other solutions which are based on injecting noise in the cache side channel are able to significantly reduce the amount of information an attacker can gather on the victim's cache state. Mukhtar et al. [53] implemented a series of independent threads that run flush and prefetch instructions during the execution of an RSA algorithm. These instructions cause random memory access patterns in the cache, obfuscating any real accesses by the algorithm from the attacker. However, their solution causes a 10% increase in the algorithm's execution time and is only applicable to this specific cryptographic algorithm and specific processors that support certain types of clflush and prefetch-like instructions. Fang Jiang et al. [54] worked at the hardware level, inserting a custom prefetcher inside the core's Dcache, which observes the number of misses inside the cache's MSHR (Miss Status Hold Register) as an indicator for a potential attack. Whenever it notices suspicious behaviour the custom prefetcher proceeds to reduce the victim's cache footprints by prefetching lines that might have been evicted by the victim back to the cache. In addition, it prefetches additional cache lines, which create more victim-irrelevant cache accesses, making it harder for an attacker to distinguish between cache accesses by the victim and the prefetcher. Their solution can result in better performance for certain benchmarks thanks to cache lines loaded by their prefetcher, while also causing performance degradation in situations where those prefetched cache lines are useless. On average, they show a 1.64% improvement in performance while incurring only a 1.26% overhead in hardware resources.

Almost all SCAs require precise timing measurements to operate correctly. Martin et al. [55] therefore modified the timing instruction RDTSC, a key component in measuring time during SCAs, to obfuscate any timing information an attacker could gather. They accomplished this by modifying the processor at the microarchitectural level, injecting noise to the values the TSC register returns, and delaying the execution of the instruction. Additionally, they modified the cache to detect accesses which could be part of an attacker's attempts to use a virtual counter. With these two modifications, they are able to show that it is impossible to distinguish between a cache hit

and miss when using these timing measurements. However, no real SCA scenarios were implemented or tested for the effectiveness of this defense. Slowdowns when using the PARSEC benchmark could range from 14% to 29% with a geometric average of 4%.

Previous work has already presented the idea of reconfigurable caches as a defense mechanism against SCAs, as shown by Bandara et al. [56]. However their configurable cache isn't implemented on an FPGA, but rather on a simulated processor using the BRISC-V platform, which employs a group of small independent memory blocks that can be used as tag or data storage and achieve run-time configurability by changing their logical organization through the writing to memory mapped registers. The area overhead caused by the addition of these blocks ranges from 162% to 858%, depending on how many of them are allocated. Additionally the core's operating frequency is also reduced, from -23% to -55%, again depending on the amount of memory blocks allocated. Dai et al. [57] also proposed a configurable LLC which they implemented with the help of a hardware tuner that changes between different configurations in fixed time periods that are smaller than the amount of time required for an attacker to gather information on the cache's state. They deployed this design on a simulated microprocessor using the GEM5 simulator and tested its effectiveness by observing the cache access patterns during different cache configurations, which showed an unstable pattern that obfuscated any real accesses by the victim. Area overheads and power consumption due to the addition of the hardware tuner are less than 1%, while benchmarks even show an improvement in performance. This is due to the fact that any application they use is known beforehand, allowing them to choose the optimal cache configuration for each one. This however limits the ability to extend this solution to general purpose embedded systems with a wide variety of applications which may not be known at design time.

Having analyzed most of the mitigation techniques presented above, we can make a number of observations. The first one is that software-based techniques cause a significantly higher performance overhead compared to hardware-based ones due to the amount of modifications required to the operating system or the software application. These changes also introduce another barrier: these cyberdefense techniques cannot be applied effectively unless significant changes are made to the operating system (or hypervisor) and/or the application software. This is something that may not be feasible in a lot of cases, while it may introduce costs and development time and

effort that can be prohibitive.

Furthermore, both software-based and hardware-based attacks (to a lesser extend) often require knowledge about the specific operation of the applications and they are not applicable in general. This limits their reach and prevents them from being used universally. Last but not least, hardware-based solutions have mostly been tested on simulated systems and not proven in actual running hardware, making both their performance and associated implementation costs an estimation.

Our hardware-based approach provides a much better solution against SCAs when compared to other techniques detailed above. Our design doesn't affect the performance of the system, due to the implementation of the detection mechanism inside the processor's core. The detection mechanism works transparently, requiring no software control, and above all with accuracy. Thanks to this, we can avoid any performance overhead from unnecessary cache reconfigurations which cost clock cycles. Even when using the periodical reconfiguration mechanism which is an entirely optional additional security measure, the performance impact is confined in the range of 1-4%.

Equally important is the fact that since our detection and mitigation mechanism is implemented in hardware, it is completely transparent to all software layers (system and application software). As such, our approach requires no changes on the software running on the processor and therefore can be applied without additional effort to all cases. We have been able to demonstrate this by implementing our solution on actual hardware and running on top of it unmodified versions of a linux-based OS along with applications and common benchmarks.

Lastly, the additional resources required for the implementation of our design are less than 2.5%, while the cost in power consumption is negligible. Again, these are actual implementation results on the final operating platform and not results of simulations or estimations by high level tools.

Taking all of the above advantages into consideration, our design provides a more cost-effective solution against cache-based SCAs compared to other proposed defenses, while simultaneously being able to detect and prevent the execution of any attack that employs the usage of timing instructions. It can be applied to any number of processors as long as they are operating on a FPGA platform, while the option to use a reconfigurable core for the implementation of the cache subsystem can be explored for ASICs.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

In this work we introduced the capability of a reconfigurable cache design as a defense mechanism against cache-based SCAs. We made use of partial reconfiguration, a feature of modern FPGA's, to implement a design that supports multiple cache configurations that are able to switch during runtime without interfering with the system's normal operation. We also implemented a small detection mechanism inside the core's CSR module, which counts the number of timing instructions, as an indicator for a potential SCA. With these modifications in place, we were able to successfully detect when an attack was about to launch and effectively prevent it, by switching to a different cache configuration and altering its characteristics.

An additional security measure was also proposed, which employs a periodical reconfiguration mechanism implemented entirely inside the processor's hardware level, switching between cache configurations in fixed time intervals, in case a variant of an attack is deployed which can't be detected by our method. This incurs a small performance drop in the system which is however minimal, depending on the reconfiguration period that has been chosen. It should be noted though that this security measure is entirely optional, and our defense can be effective without it.

Finally, we showed that our proposed defense doesn't affect the system's performance in non-attack scenarios and our modifications to the processor incur only a small overhead in resource usage, which is less than 2.5%. This makes it an excellent choice for the mitigation of cache-based SCAs, offering an efficient and cost-effective solution, compared to other defenses that have been proposed so far in the literature.

## 6.2 Future Work

This work can be expanded on a few aspects. First of all, the amount of cache configurations implemented can be increased. For the purposes of our work, we only implemented 2 configurations, the default and a second modified one. However, more can be added to increase the complexity an attacker will face when trying to gather information on the system's cache characteristics. Another aspect would be increasing the design's reconfiguration throughput. This would decrease the amount of downtime the CPU faces whenever a new cache configuration gets loaded. Storing the partial bitstreams in a different memory location is also another change that could be made, to avoid any possible interference from the OS. Lastly, the system's area overhead can potentially be improved by optimizing the amount of resources occupied by the Pblock region we specify during the design of the reconfigurable partition.

# References


[1] Chao Su and Qingkai Zeng. "Survey of CPU Cache-Based Side-Channel Attacks Systematic(Analysis, Security Models, and Countermeasures)". In: Security and Communication Networks, vol. 2021.12 (July 2021). URL: https://doi.org/10.1155/2021/5559552.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[4] Thomas Bewley. "Spycatcher (The candid autobiography of a Senior Intelligence Officer). By Peter Wright. New York: Viking Penguin. 1987. Pp 392." In: *Psychiatric Bulletin* 13.4 (1989), pp. 217–219. DOI: 10.1192/pb.13.4.217-a.

[5] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[6] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm21.html#RivestSA78.

[7] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654.

[8] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[9] Daniel J. Bernstein. "Cache-timing attacks on AES". In: 2005. URL: https://api.semanticscholar.org/CorpusID:2217245.

[10] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-32648-9.

[11] YongBin Zhou and DengGuo Feng. "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing". In: (2005). zyb@is.iscas.ac.cn 13083 received 27 Oct 2005. URL: http://eprint.iacr.org/2005/388.

[12] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* 8 (2018), pp. 1–27. URL: https://api.semanticscholar.org/CorpusID:4626560.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[16] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 199–212. ISBN: 9781605588940. DOI: 10.1145/1653662.1653687. URL: https://doi.org/10.1145/1653662.1653687.

[17] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross Processor Cache Attacks". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: Association for Computing Machinery, 2016, pp. 353–364. ISBN: 9781450342339.

DOI: 10.1145/2897845.2897867. URL: https://doi.org/10.1145/2897845.2897867.

[18] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapôtre, Muhammad Khurram Bhatti, and Guy Gogniat. "Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA". In: *Inf. Syst.* 92 (2020), p. 101524. URL: https://api.semanticscholar.org/CorpusID:216320817.

[20] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On Subnormal Floating Point and Abnormal Timing". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 623–639. DOI: 10.1109/SP.2015.44.

[21] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *The security impact of a new cryptographic library*. Cryptology ePrint Archive, Paper 2011/646. https://eprint.iacr.org/2011/646. 2011. URL: https://eprint.iacr.org/2011/646.

[22] Peng Li, Debin Gao, and Michael K. Reiter. "Mitigating access-driven timing channels in clouds using StopWatch". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575299.

[23] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. "Efficient System-Enforced Deterministic Parallelism". In: *CoRR* abs/1005.3450 (2010). arXiv: 1005.3450. URL: http://arxiv.org/abs/1005.3450.

[24] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. "The Last Mile: An Empirical Study of Some Timing Channels on seL4". In: *ACM Conference on Computer and Communications Security*. Scottsdale, AZ, USA: ACM, Nov. 2014, pp. 570–581.

[25] Michael Misiu Godfrey and Mohammad Zulkernine. "A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud". In: *2013 IEEE Sixth International Conference on Cloud Computing* (2013), pp. 163–170. URL: https://api.semanticscholar.org/CorpusID:12985228.

[26] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. "Scheduler-based Defenses against Cross-VM Side-channels". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 687–702. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan.

[28] Michael Godfrey and Mohammad Zulkernine. "Preventing cache-based side-channel attacks in a cloud environment". In: *IEEE Transactions on*

*Cloud Computing* 2.4 (2014), pp. 395–408. DOI: 10.1109/TCC.2014.2358236.

[29] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud". In: *USENIX Security Symposium*. 2012. URL: https://api.semanticscholar.org/CorpusID:7391988.

[30] Zhenghong Wang and Ruby Lee. "New cache designs for thwarting software cache-based side channel attacks". In: *ACM Sigarch Computer Architecture News* 35 (June 2007), pp. 494–505. DOI: 10.1145/1273440.1250723.

[31] W.-M. Hu. "Reducing timing channels with fuzzy time". In: *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. 1991, pp. 8–20. DOI: 10.1109/RISP.1991.130768.

[32] Chong Wang, Nasro Min-Allah, Bei Guan, Yu-Qi Lin, Jing-Zheng Wu, and Yong-Ji Wang. "An Efficient Approach for Mitigating Covert Storage Channel Attacks in Virtual Machines by the Anti-Detection Criterion". In: *J. Comput. Sci. Technol.* 34.6 (Nov. 2019), pp. 1351–1365. ISSN: 1000-9000. DOI: 10.1007/s11390-019-1979-8. URL: https://doi.org/10.1007/s11390-019-1979-8.

[33] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating fine grained timers in Xen". In: CCSW '11 (2011), pp. 41–46. DOI: 10.1145/2046660.2046671. URL: https://doi.org/10.1145/2046660.2046671.

[34] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." In: *IACR Cryptology ePrint Archive* 2006 (Jan. 2006), p. 52.

[35] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. "NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '18. Los Angeles, California: Association for Computing Machinery, 2018. ISBN: 9781450365000. DOI: 10.1145/3214292.3214293. URL: https://doi.org/10.1145/3214292.3214293.

[36] Valentin Martinoli, Elouan Tourneur, Yannick Teglia, and Régis Leveugle. "CCALK: (When) CVA6 Cache Associativity Leaks the Key". In:

*Journal of Low Power Electronics and Applications* 13.1 (2023). ISSN: 2079-9268. DOI: 10.3390/jlpea13010001. URL: https://www.mdpi.com/2079-9268/13/1/1.

[37] Ruby B. Lee and Forrest G. Hamrick. "National Cyber Leap Year Summit 2009 Co-Chairs ' Report". In: 2009. URL: https://api.semanticscholar.org/CorpusID:16201525.

[38] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[49] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. "Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core". In: *ArXiv* abs/2005.02193 (2020). URL: https://api.semanticscholar.org/CorpusID:218502526.

[50] Jeferson Gonzalez-Gomez, Lars Bauer, and Jörg Henkel. "Cache-Based Side-Channel Attack Mitigation for Many-Core Distributed Systems via Dynamic Task Migration". In: *IEEE Transactions on Information Forensics and Security* 18 (2023), pp. 2440–2450. DOI: 10.1109/TIFS.2023.3266630.

[51] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity". In: Feb. 2015. DOI: 10.14722/ndss.2015.23264.

[52] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters". In: *Applied Soft Computing* 49 (2016), pp. 1162–1174. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2016.09.014. URL: https://www.sciencedirect.com/science/article/pii/S1568494616304732.

[53] Muhammad Mukhtar, Maria Mushtaq, M. Bhatti, Vianney Lapotre, and Guy Gogniat. "FLUSH + PREFETCH: A countermeasure against access-driven cache-based side-channel attacks". In: *Journal of Systems Architecture* 104 (Dec. 2019), p. 101698. DOI: 10.1016/j.sysarc.2019.101698.

[54] Fang Jiang, Fei Tong, Hongyu Wang, Xiaoyu Cheng, Zhe Zhou, Ming Ling, and Yuxing Mao. *PCG: Mitigating Conflict-based Cache Side-channel Attacks with Prefetching*. 2024. arXiv: 2405.03217 [cs.CR].

[55] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 118–129. DOI: 10.1109/ISCA.2012.6237011.

[56] Sahan Bandara and Michel A. Kinsy. "Adaptive Caches as a Defense Mechanism Against Cache Side-Channel Attacks". In: *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*. ASHES'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 55–64. ISBN: 9781450368391. DOI: 10.1145/3338508.3359574. URL: https://doi.org/10.1145/3338508.3359574.

[57] Chenxi Dai and Tosiron Adegbija. "Exploiting Configurability as a Defense against Cache Side Channel Attacks". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, pp. 495–500. DOI: 10.1109/ISVLSI.2017.92.

# External Links


[3] *CVA6 RISC-V CPU*. URL: https://github.com/openhwgroup/cva6.

[19] *Meltdown and Spectre*. URL: https://meltdownattack.com/.

[27] *Cache Coloring*. URL: https://en.wikipedia.org/wiki/Cache_coloring.

[39] *Genesys 2*. URL: https://digilent.com/reference/programmable-logic/genesys-2/start.

[40] *Vivado*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[41] *Dynamic Function eXchange*. URL: https://www.xilinx.com/products/design-tools/vivado/dynamic-function-exchange.html.

[42] *CVA6-SDK*. URL: https://github.com/openhwgroup/cva6-sdk.

[43] *OpenSBI*. URL: https://github.com/riscv-software-src/opensbi.

[44] *U-Boot*. URL: https://github.com/u-boot/u-boot.

[45] *Partial Reconfiguration On Xilinx FPGAS*. URL: https://www.doulos.com/media/1171/xilinx_partial_reconfiguration.pdf.

[46] *Dynamic Function eXchange Controller*. URL: https://docs.amd.com/v/u/en-US/pg374-dfx-controller.

[47] *Dynamic Function eXchange Controller*. URL: https://www.xilinx.com/products/intellectual-property/dfx-controller.html.

[48] *Dynamic Function eXchange Decoupler*. URL: https://www.xilinx.com/products/intellectual-property/dfx-decoupler.html.