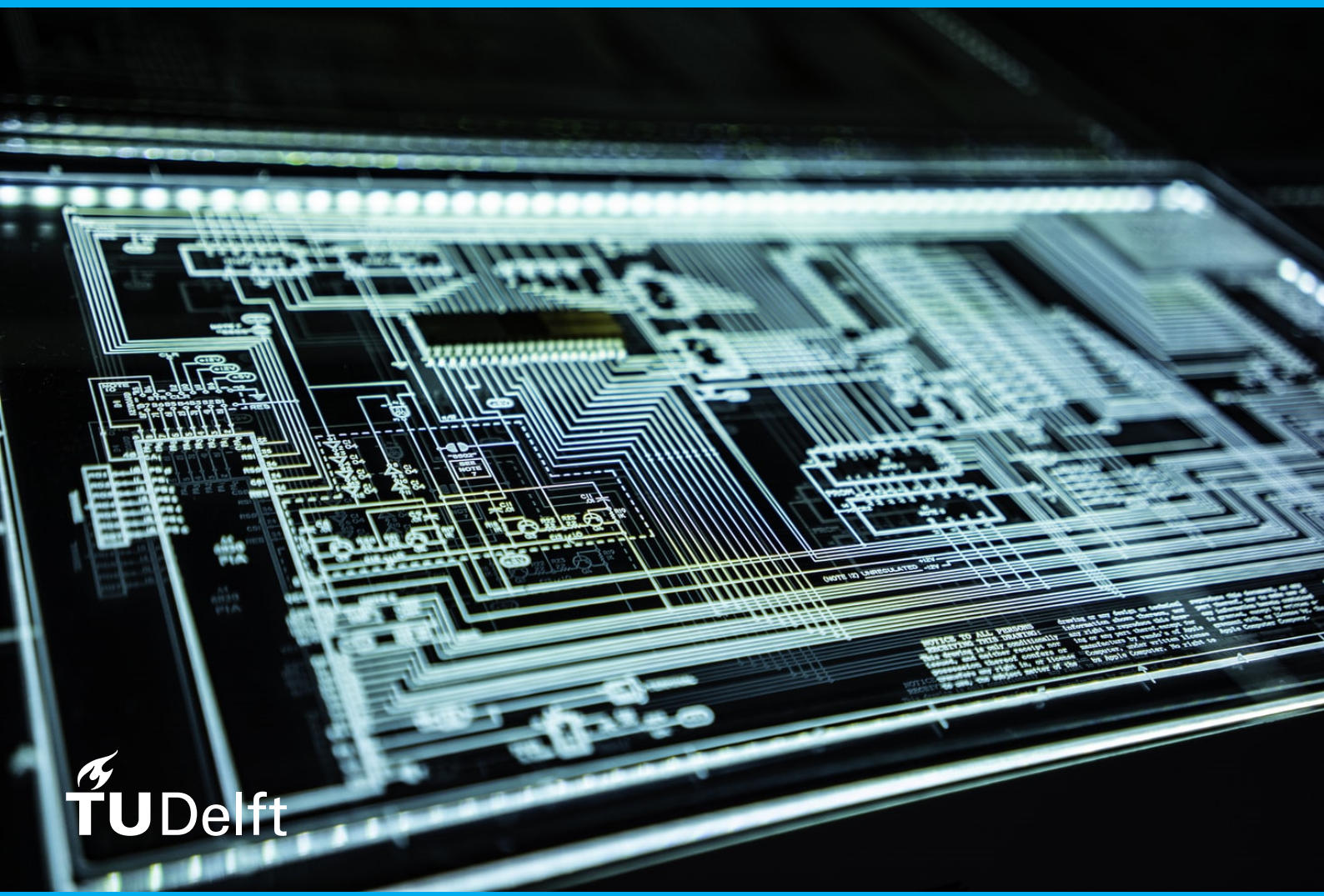
Analysis of Cache Attacks and Countermeasures on the ρ-VEX Processor

D.Verrer



Analysis of Cache Attacks and Countermeasures on the ρ-VEX Processor

by

D.Verrer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday November 23 at 10:00.

Author:	D. Verrer	
Student number:	4579801	
Thesis number:	Q&CE-CE-MS-2021-10	
Project duration:	9 November, 2020 – 23 November, 2021	
Thesis committee:	Dr. Ir. J. S. S. M. Wong	TU Delft, supervisor
	Dr. Ir. M. Taouil	TU Delft
	Dr. Z. Erkin,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Cover image from [1].

TU Delft

Abstract

Modern implementations of encryption algorithms on CPU's that use frequent memory lookups of precomputed functions, are vulnerable to Cache based Side-Channel Attacks. The ρ-VEX processor, a runtime reconfigurable VLIW processor developed at the Computer and Quantum Engineering department at the TU Delft was identified to possibly allow for special countermeasure implementations against Cache Attacks, because of its unique Cache architecture, hardware context switching and timing behaviour.

In this thesis, the possibility to use the runtime reconfigurability against Cache based Side-Channel Attacks of the ρ-VEX processor is investigated. The ability for the ρ-VEX to alter its execution time based on configuration size, interfere with its own Cache state based on configuration and dynamically switch contexts between Caches are explored.

Simple variants of the attacks Prime+Probe, Evict+Time and Final Round Collision Attack are implemented and ran on two setups that simulate practical scenarios of the ρ-VEX. These are a standalone setup and a setup with a second context sharing the processor with an arbitrary workload based on the PowerStone benchmarks, which causes noise by itself and can amplify countermeasures. The ρ-VEX was instanced on the Genesys2 FPGA development board.

We have shown an implementation of timing noise through configuration size variations called n-Lane increased the amount of samples required for the timing Attacks Evict+Time and Final Round Collision to around 800x more traces. An implementation of access noise through swapping contexts called CacheSwap achieved 800x more traces for Evict+Time and 225x more traces for Final Round Collision when executed in a shared processor. The effect on Prime+Probe was only strong for higher chances to swap, but the overhead for these percentages was considered too high. An implementation of isolating lookups over private Caches, called ScatterRound, had additional benefits aside from preventing collisions. It made our Evict+Time Attack take 160x, Prime+Probe Attack 175x and Final Round Collision Attack 225x as many samples to be successful. We have shown that the overhead associated with 10% n-Lane and 10% CacheSwap was reasonable, but ScatterRound was concluded to require a specific execution setup to achieve performance costs that are acceptable.

Acknowledgements

As one of the people who made their Master Thesis during the Covid-19 pandemic, I am happy to finish this thesis. Being forced to work from home and face all practical obstacles that came with it was an unique experience. I would like to thank the people who made it possible for me to make it all the way through the process.

I would like to thank my daily supervisors Stephan Wong and Mottaqiallah Taouil, who not only helped me through the technical process of this thesis but also motivated me, provided useful feedback to improve this work and gave me moral support if certain obstacles or delays appeared along the way.

I would like to thank Cezar Reinbrecht for getting me started on the implementation aspects of Cache Attacks, and providing a lot of useful ideas and suggestions for the project.

I would like to thank my friends and people close to me during the months I spent working on this Thesis.

Finally I would like to thank my parents and family, who were almost more excited than me for the whole process of finalizing the MSc Computer Engineering with this thesis.

Daan Verrer
Delft, October 2021

Contents

1 Introduction

This thesis describes the MSc thesis project that researches the possibility to use the ρ-VEX processor's design time reconfigurability to implement Countermeasures against Cache based Side-Channel attacks against the AES-128 encryption algorithm.

First, the context of this project is described in Section 1.1, then a brief summary of the State of the Art of Cache Attacks and Countermeasures is given in Section 1.2. In Section 1.3 the problem statement and chosen methodology is stated. Finally, in Section 1.4 the structure of this thesis is described.

1.1. Context

Encryption algorithms are designed to be logically secure. This means that while observing the input to output behaviour when encrypting a plaintext, without knowledge of the secret key, it is impossible for an attacker to come up with a method to derive the the secret key in feasible time. When these encryption algorithms are automated in logical circuits in integrated circuits, measurable behaviour of the device can correlate with the data that is processed in the device and these implementations become vulnerable to so called Side-Channel attacks that analyze this behaviour. One of these Side-Channels is the Cache Memory in a modern CPU. The Cache Memory acts as a temporary copy of data in the Main Memory. Depending on the state of the Cache and the addresses of Memory accesses, the execution behaviour of the encryption process and other processes sharing this Cache measurably change. Measuring and analyzing these execution statistics can reveal the encryption key used.

The ρ-VEX processor is a research project at the Computer and Quantum Engineering department at the TU Delft. The project is part of a larger research direction of Liquid Computer Architectures: computer hardware that can adapt to the executed workload by redistributing or changing its resources during runtime. The ρ-VEX processor is a VLIW processor that can split itself up into smaller execution cores at runtime, if the current processes running on it can not fully utilise the full instruction bundle length available in the VLIW Architecture. This reconfigurability increases the system throughput by making resources available to other processes, if these resources are measured to be underutilized by the current workload. The ρ-VEX processor is currently designed with both the option for it to be a standalone processing system, as well as a dedicated signal processing coprocessor. In both scenarios, it is possible that the ρ-VEX is utilised to do encryptions or other vulnerable secure operations, and can thus be attacked with a Cache-based Side-Channel Attack.

Because the ρ-VEX has a unique Cache design that can redistribute itself over hardware contexts running on the processor and because the reconfiguration can alter the timing and logical behaviour of the processor during runtime, the ρ-VEX was identified to find potential use in the implementation of Countermeasures against the Cache Side-Channel Attacks.

1.2. State of the Art

The notion of the CPU Cache being a potentially vulnerable Side-Channel to be exploited to retrieve secret keys from encryption algorithms was first made by Kocher [2] and Kelsey et al. [3] in their studies towards Timing Analysis Attacks. The first real Cache Attack study was described by D. Page against the DES encryption algorithm [4]. They used the timing of encryptions and knowledge on how this timing should correlate with Cache behaviour in order to describe an attack that was able to retrieve the secret encryption key. They also made the first suggestions for Countermeasures against the vulnerability in the Cache and analyzed the potential to use Cache performance traces to perform even better attacks [5]. Future work then showed practical attacks that targeted the table lookups of the AES encryption algorithm in [6] [7] [8]

A third channel for Cache Attacks, the Access Based Side-Channel, was introduced by Percival [9], in an attack against the RSA encryption algorithm. Then later, Access Based attacks were demonstrated against AES by Osvik et al. [10]. In the same paper, Osvik et al. also demonstrated the possibility for asynchronous attacks: attack where the attacker does not need explicit knowledge on exact isolated encryption samples with known plaintext/ciphertext, but could carry out attacks for random sets of input plaintexts with statistics on occurrences of plaintext symbols.

As modern systems became larger and faster, these Cache Attacks got adapted to function on these new systems. Multicore systems with shared Cache levels and shared software libraries were shown vulnerable to Access Based Cache Attacks across isolated cores, and new attacks based on these new features were proposed and shown to be effective [11] [12] [13]. Attacks across isolated virtual machines running on the same system have also been shown to be possible [14] [15] and also shown effective against AES [16].

The design of Countermeasures against the different kinds of Cache Attacks is a widely studied topic. Full redesigns of the Cache have been proposed [17] [18]. These Architectures can dynamically change the mappings between Memory addresses and Cache lines. This ambiguity for the mapping of the Memory addresses make it infeasible perform Access Based attacks. Another broadly studied approach is to implement a detection scheme [19] [20], and make the system deny services or destroy the Cache state when an attack is detected. For specific vulnerable algorithms, like the AES encryption algorithm, different implementations that bypass the use of table lookups from Memory have been proposed to make the algorithm resistant to Cache Attacks. Major implementations are bitslicing [21] [22], where multiple plaintexts are shuffled with each other and multiple encryptions are done in parallel. Another approach is to implement algorithm specific hardware in the processor. CPU manufacturers like Intel embed special hardware that implement the rounds of the AES encryption and decryption process [23].

1.3. Problem statement and Methodology

The main problem statement in this thesis is:

Can the Runtime Reconfigurability of the ρ-VEX processor be used to implement efficient Countermeasures against Cache-based Side-Channel Attacks?

In order to design Countermeasures on the ρ-VEX, first a understanding of what types of Cache Attacks exist is required. Knowledge on how these attacks function, and if they are implementable on the ρ-VEX is needed. This leads to the first subquestion, followed with the chosen approach:

1. What fundamental techniques are used in Cache Attacks

- Do a broad literature study, in order to quantify what types of Cache Attacks there have been studied, and how these can be classified.
- Present the types of classifiers found in literature, and how these can be relevant in our study.
- Study Cache Attacks belonging to a variety of these classifiers, and fully analyze those that are implementable on the ρ-VEX hardware.

From the study, a selection of attacks that we focus on needs to be made. It is important to research what the attacker can use the ρ-VEX Architecture for in Cache Attacks, how effective these attacks are

and what security issues might be caused by the Architecture itself. This leads to the following sub-question and chosen approach:

2. How vulnerable is the ρ-VEX design to Cache Attacks?

- Make a selection from the analyzed Cache Attacks that are practical on the ρ-VEX, to use them in the research for the Countermeasures.
- Implement the attacks on the ρ-VEX, and test if they work.
- Present the practical requirements to implement these attacks specifically for the ρ-VEX, and identify potential security threats in the design.
- Test the attacks under multiple operational circumstances, to get a indicator on how the attacks perform without Countermeasures.

The potential for the ρ-VEX to influence the behaviour related to the Cache needs to be analyzed, and discussed relative to existing Countermeasures that are used to prevent Cache Attacks. This results in the following sub question and methodology:

3. How can the ρ-VEX Architecture be utilized against Cache Attacks?

- Investigate the potential of the ρ-VEX, and how Cache or Cache Attack related behaviour can be influenced utilising the runtime reconfiguration.
- Discuss Countermeasures found in literature, and how these could be implemented with ρ-VEX Architecture specific functionality.
- Design Countermeasures based on these ideas, and discuss on what attacks they could have an effect.

The most important aspects on Countermeasures is how they perform against the baseline attack without Countermeasures. This can result in either preventing an attack fully, or making the attack require extra samples to perform a successful attack.

4. How effective and efficient are Countermeasures based on reconfigurability of the ρ-VEX Architecture?

- Implement the designed Countermeasures with multiple variations if applicable.
- Profile the Countermeasures on effectiveness and performance.
- Test the most likely candidates for practical implementation on effectiveness over large sample sizes, and if they can fully prevent the different attacks.
- Discuss and test how the Countermeasures could be worked around by the attacker, and if this can be prevented by the implementation.

1.4. Thesis overview

In Chapter 2, background information required to understand the rest of this thesis is given. We give a brief introduction on Memory and Cache Architectures in Section 2.1. We then introduce the ρ-VEX processor Architecture in Section 2.2, and mainly focus on how its Runtime Reconfigurability and Cache Architecture is implemented. Finally, we will describe the AES algorithm and its implementation in detail in Section 2.3.

Chapter 3 contains a literature overview of Cache Attacks and its Countermeasures, and relates this to the ρ-VEX and its potential for Countermeasures. Section 3.1 briefly explains how the implementation of the Cache can lead to Side-Channel vulnerabilities. Section 3.2 gives an overview of Cache Attacks found in literature, and identifiers used to classify them. Section 3.3 describes a set of Cache Attacks that are implementable on the ρ-VEX in detail. Section 3.4 provides an overview of Countermeasure designs against Cache Attacks found in literature. Section 3.5 introduces ideas on how to utilise the

ρ-VEX reconfigurability to implement Countermeasures, based on the Countermeasures studied in the previous section. Finally, in Section 3.6 a decision is made on what attacks from Section 3.3 to implement in order to test the potential of Countermeasures on the ρ-VEX.

Chapter 4 describes the implementation of the studied Cache Attacks on the ρ-VEX, and the design and implementation of three Countermeasures based on the ideas presented in Chapter 3. In Section 4.1 practical details regarding the ρ-VEX Architecture that are relevant for the attack and Countermeasure implementations are discussed. In Section 4.2 the implementations of the three Cache Attacks, Prime+Probe, Evict+Time and Final Round Collision are described. Then, the design and implementation of the three Countermeasure designs are described. Section 4.3 describes the Countermeasure where configuration size variations are done to cause noise in the timing measurements, called *n-Lane*. Section 4.4 introduces *CacheSwap*, where executing hardware contexts briefly swap between Caches. Finally Section 4.5 describes *ScatterRound*, where vulnerable rounds of the AES algorithm are split out over multiple Caches to prevent internal Cache Collisions.

In Chapter 5, the experimental results of the Cache Attacks against the Countermeasures introduced in Chapter 4 are presented. Section 5.1 describes the experimental setup, synthesizing the ρ-VEX processor on the Digilent Genesys2 Development board. Section 5.2 analyses the raw performance of our OpenSSL AES implementation port to the ρ-VEX and its speedup due to cached lookups. Section 5.3 gives an overview of the metrics that make up the results on the performance of the Countermeasure against the attacks, and Section 5.4 gives and overview of the experiments that are done in the rest of the chapter. Section 5.5 gives the baseline results of the attacks. Section 5.6 does experiments on the influence of the shared Memory bus between multiple processes in a shared processor. Section 5.7 gives the experimental results of the n-Lane Countermeasure, Section 5.8 that of the CacheSwap Countermeasure and Section 5.9 of ScatterRound. Finally, a conclusion on the results is given in Section 5.10

In Chapter 6, the thesis is concluded. A summary is provided, the main contributions and answer to the problem statement are formulated. Finally, suggestions for future work based on the results of this thesis are presented.

2

Background

In this chapter, background information required to understand the rest of this thesis is given. We give a brief introduction on Memory and Cache Architectures in Section 2.1. We then introduce the ρ-VEX processor Architecture in Section 2.2, and mainly focus on how its runtime reconfigurability and Cache Architecture is implemented. Finally, we will describe the AES algorithm and its implementation in detail in Section 2.3.

2.1. Memory Hierarchy and Cache Architectures

In this section a brief introduction to the Memory Hierarchy is given. We will then highlight the most important level of the hierarchy; the Cache. We will explain the structure of the Cache, and how it interacts with the Main Memory level.

2.1.1. Memory Hierarchy

The Memory of a modern computer is a hierarchical system. Multiple levels of increasing capacity and decreasing access speed are connected to each other to overcome the capacity versus performance trade off.
A schematic of the Memory Architecture, with indicators for the access delays and capacity sizesn can be seen in Figure 2.1. The general structure contains the processor itself (data registers), the processor Caches (SRAM), the Main Memory (DRAM), the hard drive (SSD) and finally slower local drives and the network.

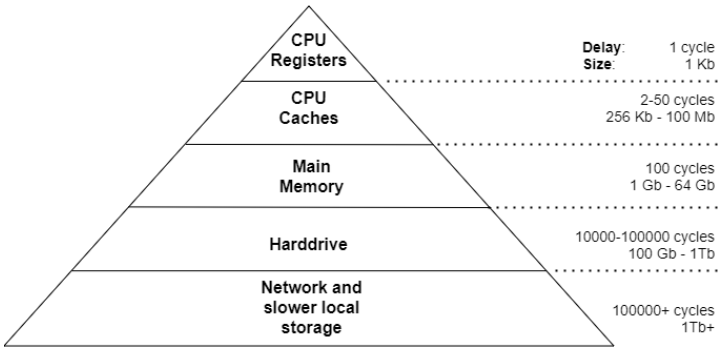


Figure 2.1: The Memory Hierarchy, with indicator values for the access delays and sizes of the Memory levels

In this Memory Hierarchy, the data on the lowest level can be (temporarily) copied to a higher level. A program can load pages of data on the hard drive to the main memory to start working with this data. Recently used data from the memory gets temporarily stored in the Cache Memory. Once the data needs to be used, the data is copied into the processor registers. These levels have minimal transaction sizes, where for instance the data copied from main memory to Cache is in chunks of 512 bits, or the memory is divided into chunks that are divided over the processes in the processor. Once results

are produced that need to be stored in the memory, this data is written down the hierarchy, depending on how long it needs to be stored.

This hierarchical structure utilises the principals of locality within computer programs and the data that allows the hierarchical system to work with this temporary copy system. This works on the higher levels, as processes that are running on the system are kept in the main memory by the operating system. This also works on the lower levels, where for instance an image is edited and the pixels are iterated over multiple times in rapid succession. This can be divided into two types of locality:

- **Temporal Locality:** If a data item is used by the processor, it is very likely it will be used again short after.

- **Spatial locality:** If a data item is used by the processor, data that is stored on addresses near that item is likely to be used soon after as well.

2.1.2. The Cache

The level of the memory right before the processor is the Cache. This Cache can be implemented as a multi layer system, where Caches of increasing size but increasing delay are connected to each other. Figure 2.2 illustrates an example Cache Architecture of a processor with two cores. The first level Cache, closest to the processor is called L1(level 1) Cache. After that can be L2, L3 etc. The higher levels of the Cache are often shared among cores or processors, depending on the system, while the lower level(s) are private to a single core.

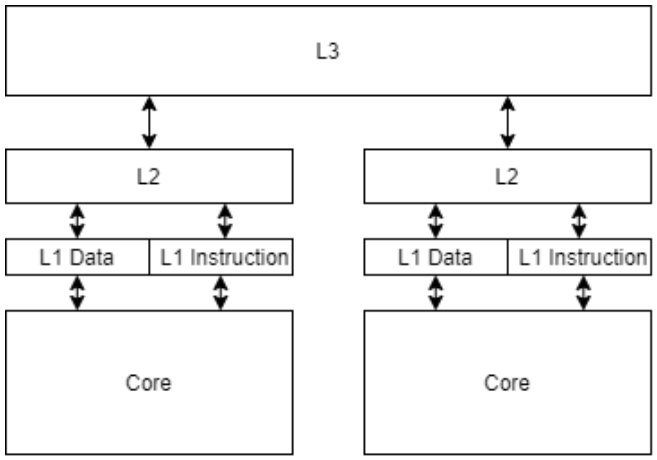


Figure 2.2: Example of a Cache Hierarchy. Level 1 is private and split between data and instruction Cache. Level 2 private and mixed data types. Level 3 is shared and mixed data types.

In practice, processors either use a single unified Cache, or divide their Caches into a separate Instruction Cache and Data Cache with their own paths to the memory. These Caches can have different implementations and specifications from each other. It is a common practice that in multi layer Caches, the lower levels are separated between instructions and data, but the higher levels are shared between the two data types.

The structure of the Cache

Figure 2.3 shows the structure of a basic Cache design.The Cache consists of lines that carry temporary copies of memory data. These Cache lines are restricted in what memory addresses can map to what line. We will look at a simple example of this mapping, the Direct Mapped Cache. When translating a main memory address to a Cache address, the address is divided into the tag field, Cache index and byte offset.

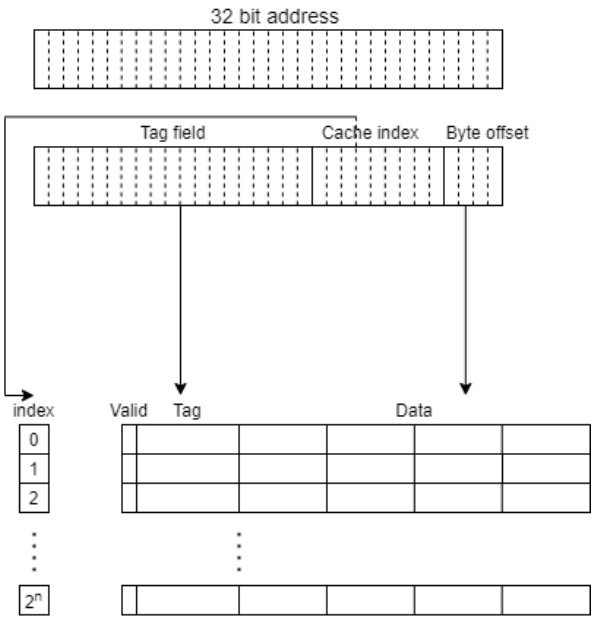


Figure 2.3: The Cache structure. With first the division of the address into the specific fields of tag, Cache line and byte offset, and then how these fields map to the specific parts of the Cache lines.

A Cache consists of a list of Cache lines. The amount of Cache lines in a basic implementation is a power of 2. Each Cache line caries the following fields:

- **Data:** the memory data that is currently copied to the Cache. This field is a multiple of the minimal data length in the processor, the processor word size. A common size of the data field in processors is 512 bits, or 16 words of 32 bits. This size is usually based on the most efficient use of how the DRAM main memory presents its data to the processor. The line used is indicated by the Cache index part of the address. What byte or word is specifically accessed within that line is determined by the byte offset field.
- **Tag:** The tag field is used to identify from what address the data on the Cache line comes. Part of the memory address is already conveyed based on the location of the data in the Cache (line + byte offset), so this field is used to carry the variable part to indicate what specific memory address is currently using this line.
- **Valid:** A bit to indicate whether the current data in this Cache is still valid; if it is the correct data that is also present in the memory on this address. Data is invalid when the system boots, and can become invalid if for instance multiple execution sources have access to the same data (Multicore systems).
- **Index:** Each Cache line has a index.It is used to identify the Cache line and used to determine what data is mapped where.

Using the Cache
The general process in a Cache when a memory lookup is requested is as followed:

1. The processor issues a memory lookup to an address a_0 e.g. 0x000000F0.
2. The Cache receives the request and does the following:
    - The Cache takes the Cache index, to see what Cache line needs to be used.
    - The Cache checks if the tag stored, and thus the current data stored, matches with the tag field of the lookup a_0.
    - If the tag matches, the Cache checks the valid bit to see if the data is valid.

3. If that tag matches, and the valid bit is correct, the byte offset is used to determine what data is forwarded to the processor.
4. If either the tag does not match, or the data is invalid, the Cache starts a transaction with the Main Memory. The Cache requests the data corresponding to the current tag field and Cache index, and thus to fill the Cache line with the correct data.
5. When the transaction is complete, the tag and valid bit are set, and the requested data is forwarded to the processor.

Cache Associativity
So far, we have discussed a Direct Mapped Cache: every Memory address maps to only one Cache line, based on the Cache index in the address field. If an address can be mapped to more than one Cache line, we speak of an set-associative Cache. Although a fully associative Cache, where every address can map to any Cache line, is only practical for very small Caches, set-associative caches are practical to implement in larger Caches. A set associative Cache means that instead of one entry per Cache index, the Cache can have multiple. This is demonstrated here with 2 entries (2-way) but can be extended to larger variants such as 4-way or 8-way Caches.

Figure 2.4 shows the new structure when going from from a 1-way associative, directly mapped, Cache structure to a 2-way Cache. We let twice as much data ”compete” for the same spots. We get more flexibility as now the same data can be placed in two different locations. This implementation requires more hardware: both to handle a lookup that can come from two lines, as well as extra bits of information to use with replacement policies. How many additional hardware we require depends on how replacement is handled.

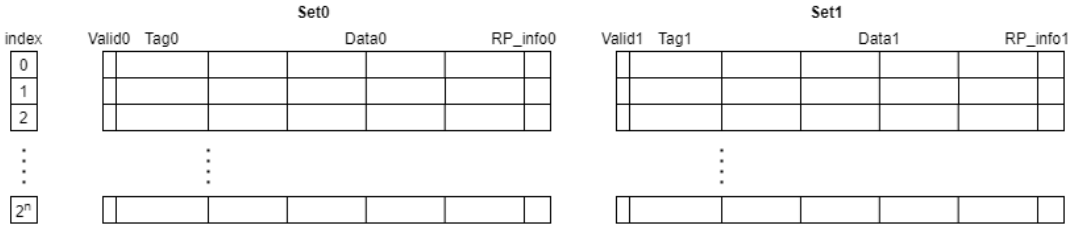


Figure 2.4: The structure of a 2-way Cache. Valid, Tag and Data fields are duplicated. Special fields for the Replacement Policy are included. Depending on the replacements policy, this could be only one field per index.

The general steps added when multiple Cache lines share a index are:

1. See if there is an empty space for the current requested Cache index (can be indicated by the valid bits) and place it in the first set.
2. If there are no empty spots left, apply a replacement policy to decide in what set the line at the current Cache index is replaced with the new data.

Replacement policy
Whenever the Cache at a certain index in a set-associative Cache if completely filled, a decision needs to be made which one of the currently occupied Cache lines to evict in order to make room for the new request. The method we use to determine this, is called the Cache Replacement Policy. Some examples of Cache Replacement Policies are:
Random Replacement: The set to replace on a Cache miss is randomly selected between the n-sets available for the address.
Least-Recently-Used (LRU): the order in which the data in the Cache is accessed is constantly kept track of. If an eviction needs to be made on a Cache line, then the oldest set is evicted and replaced with the new lookup.
Most-Recently-Used (MRU): like with LRU the order of access is also constantly tracked, but now the most recently used item gets evicted instead of the oldest one.
First-In-First-Out (FIFO): In a FIFO Replacement Policy, the order of which the data accessed entered the Cache is also tracked. Unlike with LRU and MRU, additional accesses will not ’refresh’ data. A pointer cycles over the sets and assigns what set can be evicted on the next miss.

2.2. The ρ-VEX Processor

The ρ-VEX processor is a VLIW processor developed at the Computer and Quantum Engineering department at the Technical University Delft, the Netherlands [24] [25]. It is a project developed by mostly (master) student projects and PhD candidates. The design is based on the HP VEX example [26]. The ρ-VEX processor deviates from this design with three unique main characteristics:

- The entire toolkit is **open source** and documented such to allow for academic work on the project.
- The processor is **design time configurable** on key characteristic parameters, and the toolkit is also configurable to support these parameters (HDL designs, compiler, headers etc.).
- The processor is **runtime configurable**, to adaptively switch between one large VLIW processor, or split its resources to form multiple parallel VLIW cores.

The runtime reconfiguration of the processor allows to efficiently swap between functioning as one large VLIW core, or multiple smaller parallel VLIW cores. This switching is done in order to extract as much available speedup from the two levels of parallelism available in the current workload: The Instruction Level Parallelism (ILP) of the individual processes, and the threat level parallelism in how many subtasks currently need to be completed. A scheduler on this system can potentially use this configurability to either use execution lanes that are not used in programs with lower ILP more efficiently, or to give processes with higher priority access to all resources to speedup execution.

2.2.1. VLIW processors

A Very Long Instruction Word (VLIW) Architecture [27] allows multiple computer instructions to be executed in parallel in the same processor core. The processor is implemented via separate execution lanes, which are essentially just copies of a single instruction processor pipeline. Separate instructions are bundled together by the compiler, forming one long instruction word or instruction bundle. These instructions are all submitted at the same time and execute in their own hardware components.
The relative, expected speedup of a VLIW processor is however not the perfect n times for n instruction lanes. A VLIW processor faces the following limitations from achieving its optimal speedup:

- **Limited ILP**: If there are many data dependencies between the instructions of a program, this can limit the amount of instructions that can be executed at the same time, as the next instructions are depending on instructions not yet issued. The compiler tries to achieve maximal ILP during compile time. VLIW processors have the possibility to issue NOP operations in lanes that get no instructions from the current bundle.
- **Limited resources per lane**: In practical processors, not every lane has the same execution units and thus not every instruction can be issued to every lane. Common instructions limited to only a couple lanes are floating point operations because of their huge hardware costs, branching units because only one branch is possible at once and Memory lookups, as the amount of parallel lookups can be limited by Cache and Memory implementations.

2.2.2. The ρ-VEX Architecture

Figure 2.6 shows a schematic of the ρ-vex hardware involved with the reconfigurability in a 8 lane, 4 context ρ-vex.

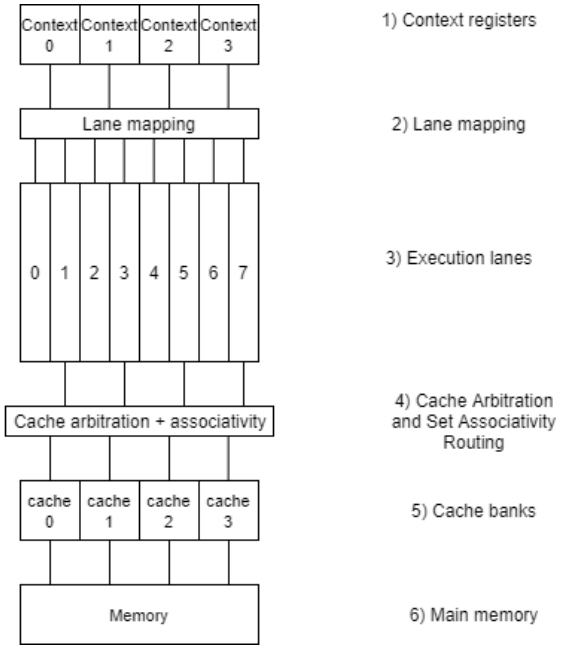


Figure 2.5: The main components of the design time reconfigurable ρ-VEX

The ρ-VEX Architecture consists of the following elements:

1. **context registers:** hardware contexts with register files containing all current data on the execution of that context. Including data registers, branching related registers and context specific control registers. Makes it possible to make multiple hardware processes to run in parallel.

2. **The lane mapping:** connecting the execution lanes to the right context depending on the configuration of the system

3. **The execution lanes:** grouped together in lanegroups of 2 lanes. The hardware in every lanegroup can differ, but at least all basic hardware functionality must be present in a single lanegroup to support the smallest configuration size.

4. **Cache arbitration + accociativity routing:** The Caches can be grouped together to form set-associative Caches. The logic that handles this mapping based on the current system configuration is included in this hardware.

5. **Cache banks:** the physical data and instruction Caches. These are connected to the instruction lanes via the previously mentioned routing system to form a L1 Cache.

6. **Memory:** The Main system Memory.

The ρ-VEX processor can reassign its execution lanes between the 4 hardware contexts. This goes in multiples of at least two lanes, which is called a lanegroup. We are also limited to cores with a power of 2 execution lanes. Figure 2.6 shows the possible combinations of configuration sizes in a 8-lane ρ-VEX, These are 1x8, 2x4, 4x2 and 1x4 + 2x2. Lanegroups can also be fully disabled for power saving purposes. If for instance a context rarely has the ILP required for performance gain when reconfigured from 4 to 8 lanes, then the context can be configured to only use 4 lanes and disable all other hardware resources to save power.

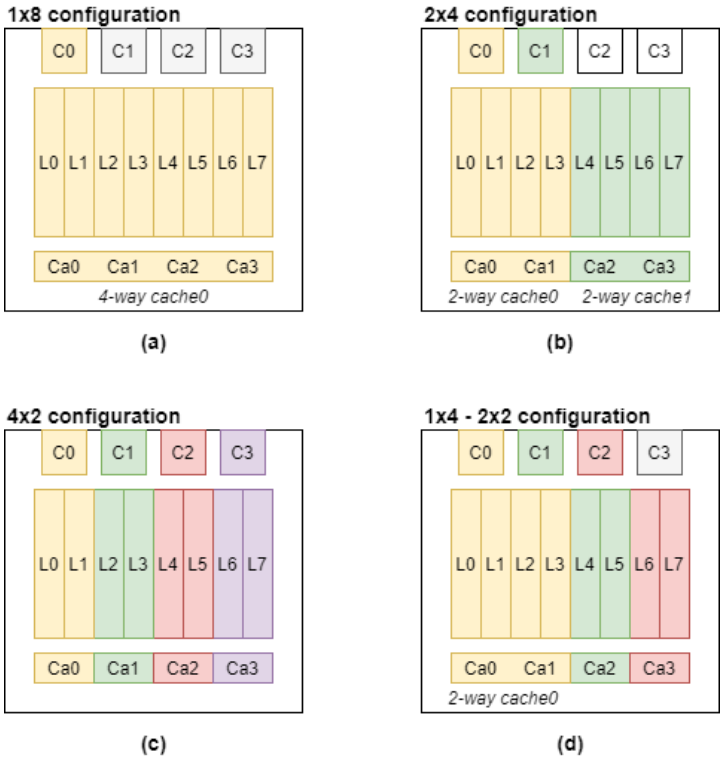


Figure 2.6: The configuration options in a 8-way ρ-VEX, showcasing how the hardware contexts C_n, instruction lanes L_n and Caches Ca_n are grouped together in these configurations

The ρ-VEX implements so called generic binaries [28] to make the compiled instruction bundles executable in each configuration possible. If these were not implemented, then each program would be compiled three times for all possible issue widths, and take near three times the amount of Memory to store. Different versions then also need to be swapped for each other in the Cache after configuration changes, resulting in large performance costs when swapping between configuration sizes.

Generic binaries have the following characteristics:

- The maximum instruction bundle size is the same as the maximum size of ρ-VEX processor.
- The bundle is either filled with NOP instructions if the current bundle does not have the maximum amount of instructions possible, or a stop bit is added to the final instruction, allowing the hardware to know where the bundle ends.
- If an executing context does not have the maximum amount of instruction lanes assigned to it, the instruction bundle is split up and executed sequentially.
- Because of this splitting, in order to guarantee every lane can execute all programs, every lane pair of the processor must contain the same hardware resources. Thus $L_0 = L_2 = L_4 = L_6$ and $L_1 = L_3 = L_5 = L_7$
- Only one Memory instruction and one branching instruction can be executed per bundle. This means that in the bundles only one instruction in lane L_1 can be one of these instructions. This is also means that while in 1x8 mode only L_1 gets these instructions, Lanes L_3 L_5 and L_7 still needs to support these instructions, because they are executed in those lanes in smaller configurations.

2.2.3. Runime Reconfigurability

A reconfiguration for the Runtime Reconfiguration system can be issued via three sources on the ρ-VEX:

1. Directly into code via a write to a register. Every context can write a new reconfiguration to address of its local CR_CRR register to request a new configuration.

2. Via a device connected to the USB UART debug bus, writing a new configuration to the global control register CR_BCRR register via this debug bus.

3. Via the sleep- and wake up system, using interrupts. Automatically triggering a set reconfiguration on interrupts to for instance run a default configuration to handle interrupts with a specific context, or to save power when the processor has to sit idle and wait for an interrupt.

Reconfiguration words
A configuration is encoded in a configuration word. This 32-bit word is used to describe a configuration of the ρ-VEX. Every nibble of 4 bits corresponds to a lanegroup of 2 execution lanes and its corresponding Cache.

In the ρ-VEX configuration used for this thesis, the default configuration, 8 lanes grouped into 4 lanegroups and 4 hardware contexts are present in the processor. The 4 least significant bits of the reconfiguration word are used to indicate the processor configuration. The 4 contexts are indicated with their integer index: 0..3. A lanegroup can also be disconnected if it is not used, using the value 8. Values 4..7 and 9..15(F) are not used in this setup, and configurations with these values are illegal and ignored by the system.

examples: assigning all lanes to context0 is a configuration written as 0x0000. Assigning only 2 lanegroups to this context is 0x0088 or 0x8800. Sharing the processor with all 4 contexts, context0 to context3, is 0x0123 or any variation with a different order.

There are some rules these configuration must adhere to:

1. Lanes used by the same context must be next to each other. Configuration 0x0011 is valid, but configuration 0x0101 not.

2. Lanegroups can only be assigned to a context in powers of two, assigning 3 lanegroups to a context is not possible and thus a configuration like 0x0001 is illegal.

3. The executing contexts must be ordered from largest to smallest configuration. 0x0012 is valid, 0x1002 is not.

The reconfiguration process
The main steps that are executed in hardware whenever a reconfiguration is requested are:

- **Decoding:** The requested configuration word is first verified on its validity for the current ρ-VEX hardware parameters. Arbitration that can happen when multiple sources request a reconfiguration at the same time is also handled. Only one reconfiguration is accepted, and the others are rejected.

- **Synchronization:** if a reconfiguration influences a context, the current operation must be halted and continued in the new setting. The pipeline is flushed, and the PC hardware adjusts to properly continue the instruction flow in this new configuration. Also, in this stage it is made sure that Memory operations and Cache write buffers are handled properly.

The reconfiguration overhead depends on the configuration issued, the current configuration and the state of the Cache write buffer and Memory hardware. This overhead is in the order of tens of cycles. The effective delay experienced by a context is also more if it is the context requesting the reconfiguration, as that adds the pipeline delay before the reconfiguration reached the register, and the pipeline is flushed when the reconfiguration starts.

2.2.4. The ρ-VEX Cache

The ρ-VEX processor includes L1 Data and Instruction Caches. The Caches are designed to support the Runtime Reconfigurability of the processor and can also be reconfigured to either function as a single large Cache, or as multiple smaller Caches divided over the contexts.

The instruction Caches have 8 32-bit words per Cache line, this is the maximum length of a single instruction bundle supported by the ρ-VEX. The data Caches have a single 32-bit word per Cache line.

The Caches are divided into four individual Cache banks. These Cache banks are connected to the processor via Set Associativity routing. This routing routes n of these Caches to the lanes used by a hardware context, where n corresponds to amount of lanegroups assigned to the context. The Caches are physically aligned with the lanegroups: using lanegroup 1 with lanes 0 and 1 also yields the usage of cachebank 0 to the context.

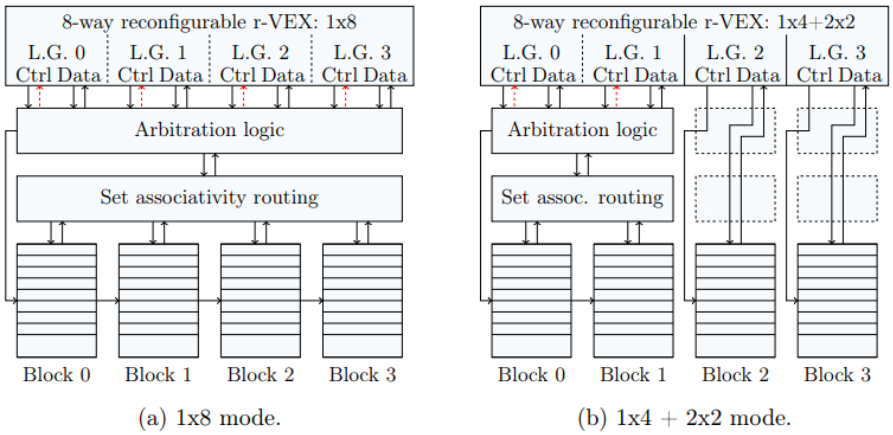


Figure 2.7: Diagram illustrating the organization of the reconfigurable data Cache. Instead of showing the generalized reconfigurable logic, the effective organization for two example configurations is shown. [29]

Whenever more than 1 Cache is assigned to a context (4-way or 8-way ρ-VEX mode), the Caches are used together as an n-way associative Cache: multiple Cache lines map to the same set address. The replacement policy used in the current implementation is to use the next bit (4 lane ρ-VEX configuration with a 2-way associative Cache) or next two bits (8 lane ρ-VEX configuration with a 4-way associative Cache) of the address to determine what Cache to use.
This practically means that after reconfiguration, the Cache data that is still present in the currently assigned Caches can be accessed if needed. As the old data gets evicted out of the Cache, the Cache starts to behave as directly mapped Cache, but now includes extra bits of the address to index Cache sets, as the Cache in use has increased in size 2 or 4 times. The n-way functionality currently only allows data cached on the ”wrong” Cache line to still be used after a reconfiguration, but because this simple replacement policy is implemented, performance gains from multi-way associative Caches are not achieved in the current implementation of the Cache. The only performance gain is that items that are currently placed in the wrong set are not ignored, but can still be used after a reconfiguration.

Memory access

In the current implementation of the ρ-VEX, only one access to the system Memory is processed at a time. This effectively means that all issued Memory accesses in other contexts have to be stalled in case a Memory Access is currently being done by another context. This means that the performance of a context depends on how much the rest of the system is shared and on the amount of Memory Accesses done by those contexts.

2.2.5. The current state of the project

Aside from the parameterised VHDL design files to instance the processor, the processor has a toolbox to support working with the processor. This includes: a compiler, a debug program to interface with the processor over a USB UART connection and VEXParse to optimize bundle sizes where the compiler fails to extract maximum parallelism. The most recent release of the ρ-vex, 4.2, is based on the redesign made in [29], which introduced the ability to reconfigure during runtime, created a reconfigurable Cache Architecture and added tracing and debugging functionality. Other recent developments on the project includes Floating Point units, ports of Operating Systems [30] and Virtual Address support with a MMU [31]. The processor has also been implemented as an ASIC chip [32].

2.3. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [33] is the cyptographic algorithm studied in this thesis. The focus is on AES because of the reliance on key dependent memory lookups during encryption in fast practical implementations on modern processors [33], and because it is widely studied in the field of Cache Attacks [10] [34] [35] [7] [36].

In Section 2.3.1 the general concept of the AES encryption algorithm is explained. In Section 2.3.2 the details of the round transformations during encryption and decryption are explained, and in Section 2.3.3 the key expansion algorithm is explained. Finally in Section 2.3.4 the T-Table implementation, the one used in this thesis, is explained.

2.3.1. The Rijndael AES proposal

The AES encryption algorithm is a blockcipher. The input plaintext data is split into blocks of either 128, 192 or 256 bits and encrypted to a 128, 192 or 256 bit ciphertext respectively. The encryption key used can also, independently of the length of the plaintext, be 128, 192 or 256 bits.
The data during encryption is represented as a 4xM matrix, where M depends on the data size of the plaintext/key in that is represented (4, 6 or 8). The first dimension, size 4, referred to the as the rows and the second dimension being the columns. See Figure 2.8 for the block notation of 128 and 192 bit plaintext blocks. The encryption key and ciphertext are represented in the same way.

128 bit plaintext

p0,0	p0,1	p0,2	p0,3
p1,0	p1,1	p1,2	p1,3
p2,0	p2,1	p2,2	p2,3
p3,0	p3,1	p3,2	p3,3

192 bit plaintext

p0,0	p0,1	p0,2	p0,3	p0,4	p0,5
p1,0	p1,1	p1,2	p1,3	p1,4	p1,5
p2,0	p2,1	p2,2	p2,3	p2,4	p2,5
p3,0	p3,1	p3,2	p3,3	p3,4	p3,5

Figure 2.8: Block form notation of the AES plaintexts

In this thesis we will focus on AES-128; encryptions with a 128 bit plaintext and a 128 bit key and consequently a 128 bit ciphertext. These are represented as a 4x4 matrix, like the left matrix in Figure 2.8. The AES encryption is implemented as n encryption rounds, described in Figure 2.9. The final round is different while all others do the same steps in order. For AES-128, there are 10 total rounds. 9 iterations of the main round and then the final round. Before the first round, an individual AddRoundKey operation is done. The output after every individual step is referred to as the state x of the encryption, and stays represented as a matrix of the same size as the input plaintext. Details on the individual steps are explained in the following sections.
The initial key used is expanded to a list of 11 roundkeys [k_0 ... k_{10}]. The key value used in the AddRoundKey steps is the next roundkey value from the previous AddRoundKey operation. The key expansion algorithm is explained in Section 2.3.3.

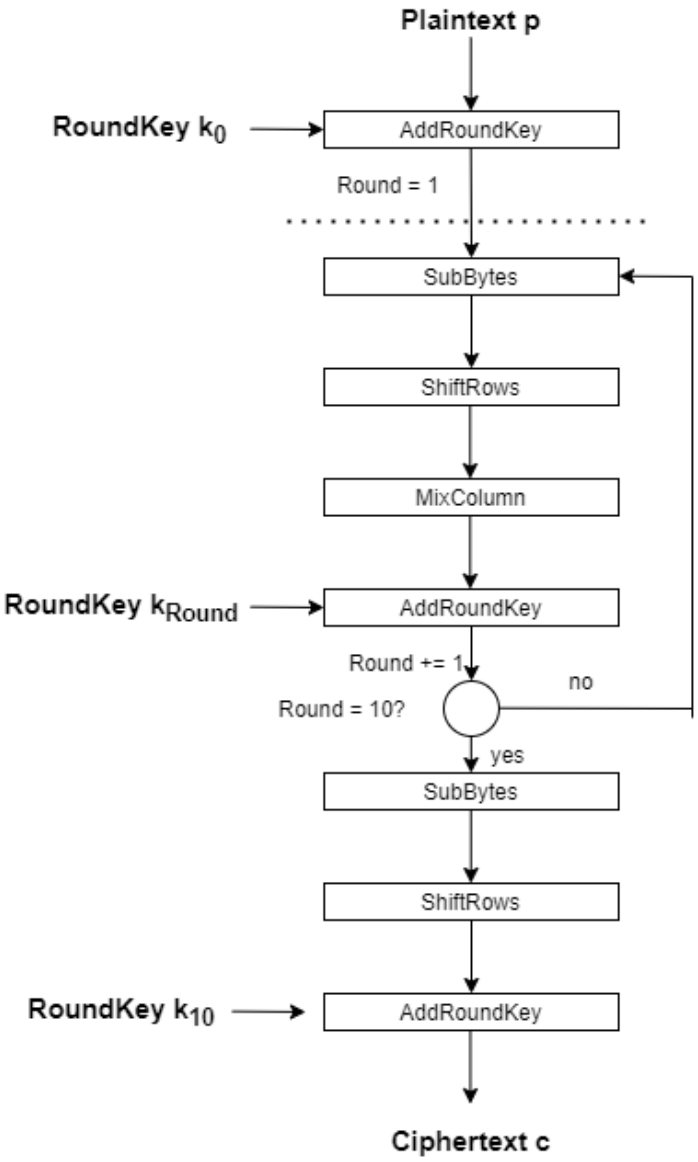


Figure 2.9: The steps of AES-128

Since AES is a symmetric encryption scheme, the decryption of ciphertexts is done with the same key via an inverted version of the encryption. Inverted versions of the respective operations in encryption are executed. We will not go into further detail on the decryption process in this thesis.

2.3.2. The AES operations

AddRoundKey

The AddRoundKey operation does a bit wise XOR between the current state *x* and the RoundKey *k*. It has new state *y* as output. Figure 2.12 visually shows what happens to the AES state represented as a block. Equation 2.1 shows the algebraic notation of the new state of column j.

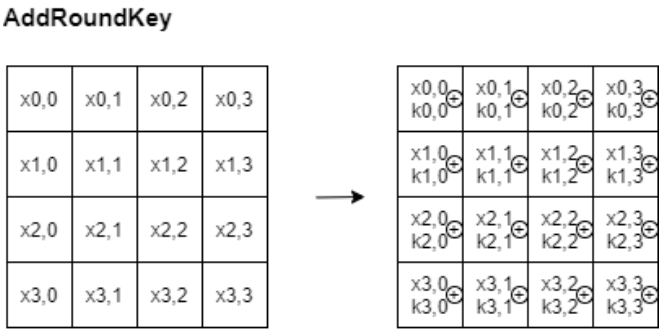


Figure 2.10: AES AddRoundKey step

$$\begin{bmatrix} y_{0,j} \\ y_{1,j} \\ y_{2,j} \\ y_{3,j} \end{bmatrix} = \begin{bmatrix} x_{0,j} \\ x_{1,j} \\ x_{2,j} \\ x_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.1}$$

SubBytes

In the SubBytes operation, every individual byte in the current encryption state is substituted with a different byte, based on the following transformation:
1) The multiplicative inverse in GF(2^8) is taken, with '00' mapped to itself.
2) an affine transformation is done according to Equation 2.2

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{2.2}$$

When implemented on modern computing systems, this transformation is precomputed and stored in a 256 byte lookup table, the so called S-box. Figure 2.12 shows the state transformation in the block notation. Table 2.1 contains the precomputed values of the S-box.

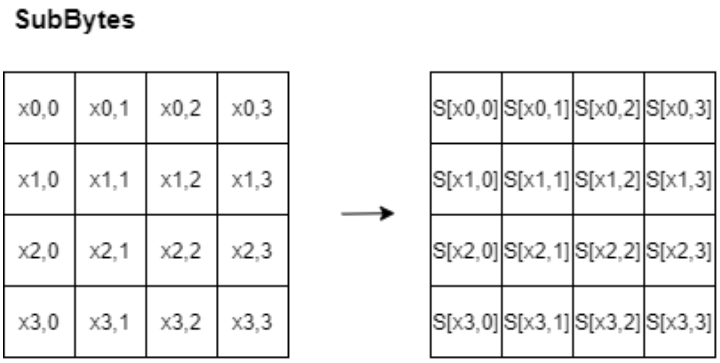


Figure 2.11: AES SubBytes step

Table 2.1: AES S-box lookup table.

	X0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0X	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

ShiftRows

ShiftRows cyclically shifts the bytes in the 4 rows of the state. The upper row is not shifted, the other 3 rows are shifted by constants C1, C2 and C3. The values depend on the plaintext block size. In AES-128 these constants are C1 = 1, C2 = 2, C3 = 3. Figure 2.12 shows the transformation on state *x* in the block notation, tracking every input column with a unique color. Equation 2.3 shows the algebraic notation of the new state of column j at the output.

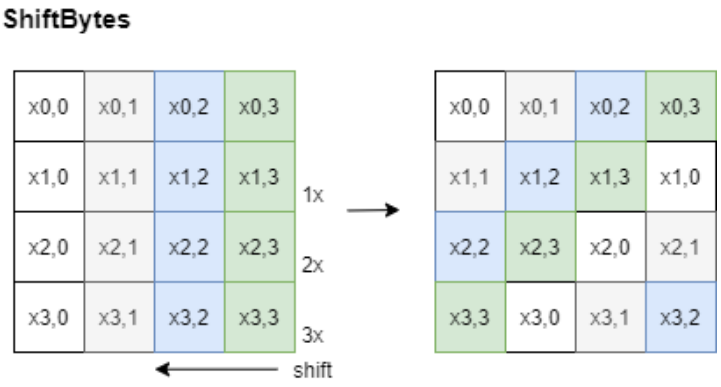


Figure 2.12: AES ShiftRows step

$$\begin{bmatrix} y_{0,j} \\ y_{1,j} \\ y_{2,j} \\ y_{3,j} \end{bmatrix} = \begin{bmatrix} x_{0,j} \\ x_{1,(j+C1)modNb} \\ x_{2,(j+C2)modNb} \\ x_{3,(j+C3)modNb} \end{bmatrix} \tag{2.3}$$

MixColumns

MixColumns is a column wise transformation. Every new element of the column is computed based on all the values in the column. The transformation is done as a matrix multiplication in GF(2^8). Figure 2.13 shows the transformation on state *x* in the block notation. Equation 2.4 shows the algebraic notation of the new state of column j at the output.

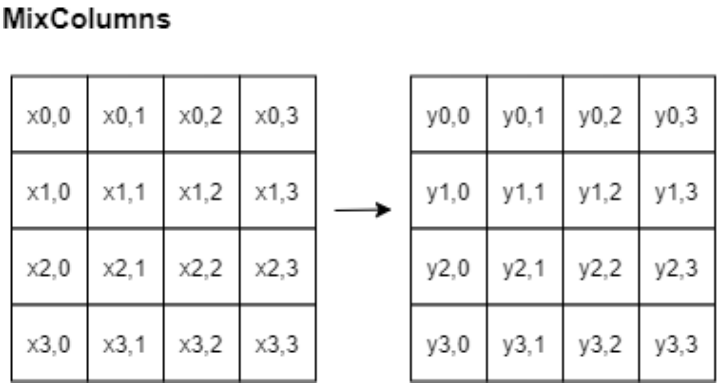


Figure 2.13: AES MixColumns step

$$\begin{bmatrix} y_{0,j} \\ y_{1,j} \\ y_{2,j} \\ y_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_{0,j} \\ x_{1,j} \\ x_{2,j} \\ x_{3,j} \end{bmatrix} \tag{2.4}$$

2.3.3. The KeyExpansion

The KeyExpansion (or key scheduling) is computation of the RoundKeys used during AES encryption. This done as a transformation of the key. Algorithm 2 describes the KeyExpansion algorithm. It takes the 16-byte key *k*, and produces 11 16-byte roundkeys k_i, stored in output W.

Algorithm 2 The AES-128 KeyExpansion algorithm as described in [33]

```
1:  KeyExpansion(byte Key[16], word W[44])
2:  for i = 0; i < 4; i++ do
3:      W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]);
4:  end for
5:  for (i = 4; i < 44; i++) do
6:      temp = W[i - 1];
7:      if (i%4 == 0) then
8:          temp = SubByte(RotByte(temp)) ^Rcon[i / 4];
9:      end if
10:     W[i] = W[i - Nk] ^temp;
11: end for
```

The SubByte operation is the same operation like discussed in Section 2.3.2. The RotByte operation is a cyclic shift by 1 word, like the bytes of the AES state in the second row are shifted in the ShiftBytes step of the AES algorithm. The Rcon (Round constant) value is the following list: Rcon = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36]

2.3.4. The 32-bit implementation

Because the individual steps of AES handle operations on a individual byte level, the performance of the AES algorithm can be rather poor modern processor architectures that are either 32-bit or more recently 64-bit architectures. For implementation of the algorithm on a 32-bit device, the original Rijndael proposal suggest the following: The steps Bytesub, Shiftrow, Mixcolumn and AddRoundkey can be written as the following equations, with $x_{i,j}$ the input state bytes and $k_{i,j}$ the elements of the roundkey:

Bytesub:

$$b_{i,j} = S(x_{i,j}) \tag{2.5}$$

Shiftrow:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,(j+C1)modNb} \\ b_{2,(j+C2)modNb} \\ b_{3,(j+C3)modNb} \end{bmatrix} = \begin{bmatrix} S(x_{0,j}) \\ S(x_{1,(j+C1)modNb}) \\ S(x_{2,(j+C2)modNb}) \\ S(x_{3,(j+C3)modNb}) \end{bmatrix} \tag{2.6}$$

MixColumn:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S(x_{0,j}) \\ S(x_{1,(j+C1)modNb}) \\ S(x_{2,(j+C2)modNb}) \\ S(x_{3,(j+C3)modNb}) \end{bmatrix} \tag{2.7}$$

AddroundKey:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S(x_{0,j}) \\ S(x_{1,(j+C1)modNb}) \\ S(x_{2,(j+C2)modNb}) \\ S(x_{3,(j+C3)modNb}) \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.8}$$

If we write these steps as one equation, we get the operation of a single AES round as an equation. Note that the mod Nb part of the Shiftrow stage is removed from this notation to keep it more compact. The equation per column becomes:

$$\begin{bmatrix} y_{0,j} \\ y_{1,j} \\ y_{2,j} \\ y_{3,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} S(x_{0,j}) \oplus \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} S(x_{1,j+C1}) \oplus \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} S(x_{2,j+C2}) \oplus \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} S(x_{3,j+C3}) \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.9}$$

These first four individual xor terms, based on permutations of S-Box lookups, can be precomputed in the following four, 1024 byte tables:

$$T_0[a] = \begin{bmatrix} S(a) \cdot 02 \\ S(a) \\ S(a) \\ S(a) \cdot 03 \end{bmatrix} T_1[a] = \begin{bmatrix} S(a) \cdot 03 \\ S(a) \cdot 02 \\ S(a) \\ S(a) \end{bmatrix} T_2[a] = \begin{bmatrix} S(a) \\ S(a) \cdot 03 \\ S(a) \cdot 02 \\ S(a) \end{bmatrix} T_3[a] = \begin{bmatrix} S(a) \\ S(a) \\ S(a) \cdot 03 \\ S(a) \cdot 02 \end{bmatrix} \tag{2.10}$$

And then finally form the computation for a column after a full AES round:

$$\begin{bmatrix} y_{0,j} \\ y_{1,j} \\ y_{2,j} \\ y_{3,j} \end{bmatrix} = T_0[x_{0,j}] \oplus T_1[x_{1,j+C1}] \oplus T_2[x_{2,j+C2}] \oplus T_3[x_{3,j+C3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.11}$$

The final round of the encryption is performed like the regular implementation of AES, since the final round lacks the MixColumn step and thus cannot use the T-tables. This final round is either implemented with four special 1024 byte tables, computing the S-Box values and placing them at the right byte offset or by using the regular S-box like described in 2.3.2 and shifting the bytes.

2.4. Conclusion

In this chapter, the necessary background to understand the rest of this thesis was given. The design of the Cache Memory, and its location within the Memory Hierarchy of a computer is briefly introduced. The focus was to clarify the general concepts of Cache implementation in order to understand how these can be vulnerable to the Cache attacks in the later chapters. Then a overview of the general concept of the ρ-VEX processor was given, with focus on understanding how the run-time reconfiguration is implemented, and how the Cache of the system is designed to work with this reconfigurability. Finally the AES encryption algorithm is described. Giving a detailed description of the steps in AES-128 in order to understand how this algorithm can become vulnerable to Cache based attacks. Finally a fast implementation of the algorithm, the T-Table implementation, is discussed as it is the most common implementation on modern systems, and is the one studied in this thesis.

3

Cache Attacks and Countermeasures literature overview

This chapter contains a literature overview of Cache Attacks and its Countermeasures, and relates this to the ρ-VEX and its potential for Countermeasures. Section 3.1 briefly explains how the implementation of the Cache can lead to Side-Channel vulnerabilities. Section 3.2 gives an overview of Cache Attacks found in literature, and identifiers used to classify them. Section 3.3 describes a set of Cache Attacks that are implementable on the ρ-VEX in detail. Section 3.4 provides an overview of Countermeasure designs against Cache Attacks found in literature. Section 3.5 introduces ideas on how to utilise the ρ-VEX reconfigurability to implement Countermeasures, based on the Countermeasures studied in the previous section. Finally, in Section 3.6 a decision is made on what attacks from Section 3.3 to implement in order to test the potential of Countermeasures on the ρ-VEX.

3.1. Information leakage in Caches

Modern computer Operating Systems allow applications to have their own isolated memory regions: only authorized processes can access data in certain regions of the memory. This is implemented via authorization systems that determine what sources can open specifics files from the hard disk, but also on a hardware level in the memory. Processes are assigned memory regions (pages) which are tracked to belong to a certain process. This memory isolation is usually implemented in a hardware component called the Memory Management Unit (MMU). This MMU maps the virtual memory addresses in programs to real physical memory addresses.
Between the main memory and the CPU is the Cache layer. The Cache functions as a temporary buffer between the memory and the CPU. Recently used data is stored in the Cache, and can be evicted once another piece of data is mapped to the same line. See Section 2.1.2 for a detailed description.

The Caches in the CPU are commonly shared between processes that execute on the same system, and do not implement physical isolation of space by default, which is the case for the main memory. The logical memory isolation rules do however also apply to data in the Cache, preventing direct access to data belonging to other processes. This isolation is just an direct extension of the main memory isolation, as the Cache shares the same access interface from the processors perspective, as if the memory was directly connected. Data in the cache can not be directly accessed by processes that it does not belong to.
However, what is not prevented is that data from two isolated memory regions share Cache lines and thus cause interaction between two processes that share the Cache. Data from multiple processes can be in constant ”fights” over Cache space. Evicting data from other processes to make room for their own data. Cache Attacks aim to abuse these interactions in the Cache, and use the following practical implementation details of the Cache:

Hit/miss execution time distinction
The Cache is used as a smaller, faster memory unit between the main memory and the CPU. If a piece of data is missing from the Cache, the processor has to start a transaction with the main memory, to load the data from the memory to the Cache, and then through the Cache to the CPU. If a line of data is missing from the Cache, the execution time of this single memory lookup uses more cycles than if the data was present in the Cache. These extra cycles are called the miss penalty. This means that measuring a single memory lookup in isolation can show directly if the data came from the Cache, or from the memory. Multilevel Caches do have different miss penalties per level, so also the level in which it is present is measurable in those systems with a certain accuracy. This also means that for a process that has memory accesses, the execution time of this process varies depending on whether this data comes from the Cache, or the memory. The general execution time correlates with the amount of hits and misses in the Cache.

Cache eviction policies and Cache sharing
In the Cache, memory data is mapped to Cache lines based on parts of the memory address. If the Cache architecture of the device under attack is known, an attacker can manipulate the data of other processes in the Cache, given the attacker understands the Cache topology. The attacker accesses enough data on certain addresses, and can analyse either the Cache behaviour during the attacked process, or can analyse its own Cache behaviour as a result of the attacked process. The Side-Channels used to observe the Cache hits/misses are discussed in Section 3.2.1. Examples of practical attacks that derive information on Cache usage by the AES-128 implementation are discussed in that section as well.

Practical Cache architecture details
Although ideally the information retrieved is that of individual memory lookups, it is not always possible to do that in the Cache architecture under attack. Modern memory chips allow for quick memory reads of groups of bits. A common resolution is 512 bits or 64 bytes burst reads, which result in multiple processor words of 32/64 bits being read at the same time, and transferred to the processor in one transaction. For efficiency , Cache lines in those CPUs are then made to store one single burst lookup, and thus end up consisting of multiple processor words at the same time. This entire line is evicted and replaced if a Cache miss occurs. This means that practical Caches that have longer Cache lines can only leak addresses up to a certain uncertainty of the last couple bits, as different accesses within a Cache line are indistinguishable from each other.

Another implementation detail is that of multi-way Caches and replacement policies. Multi-way Caches make multiple Cache lines can map to the same memory address. A replacement policy determines what Cache line will be used by a requested data line. This replacement policy influences what the attacker need to do to force the targeted data in or out of the Cache. For instance on a LRU policy, the attacker has to do multiple (n) data lookups mapping to the same set to make sure the targeted data is evicted from it, as the attacker has no knowledge on what specific set the targeted data was placed.

3.2. Overview of Cache Attacks

Cache Attacks can be classified based on different identifiers. The most common classifier is to split based on Side-Channel used to perform the attack. The three classifiers commonly used in literature are:

- **Access-driven:** Cache hits and misses are directly measured by the attacker timing individual memory accesses.
- **Time-driven:** The victim process is timed by the attacker, and correlated with theoretical Cache hits and misses on certain targeted memory accesses.
- **Trace-driven:** The pattern of Cache hits and misses are directly registered during the victim process.

In Section 3.2.1, a overview of Cache Attacks against AES found in literature grouped by Side-Channel used is given. Then, Section 3.2.2 briefly introduces other classifiers used to describe Cache Attacks, and how they can be relevant for this Thesis.

3.2.1. Cache Attacks overview: based on Side-Channel used

In Cache Attacks, patterns of Cache hits and misses are studied during, or shortly after operation of the crypthographic algorithm under attack. What Side-Channel is used to derive information on the Cache behaviour is a classifier that determines the fundamental type of samples that will be used during the attack. The Side-Channels that can be used are:

Access-driven attacks

In access driven attacks, a degree of Cache sharing happens between the victim process and the attacker. The attacker tries to either actively interfere with the Cache data used by the victim process, or lets the victim process interfere with its own data.

The simplest example is the Prime+Probe Cache Attack [10], where large tables of data are first loaded into memory, a victim process executes and after operation observes the presence of the primed data in the Cache. Missing entries correspond to evicted Cache lines and thus show the used Cache lines by the victim process.

The Evict+Time attack [10] first makes the attacked algorithm run on a chosen input to initialise all its table data used in the Cache. Then the attacker tries to evict one lookup used by the attacked algorithm and measures the time of a rerun with the same data. This measurement data correlates if a address that is accessed for this input data is evicted or not.

If an attacker has access to the same memory data via shared libraries on the system, the attacker can carry out the so called Evict+Reload[11] attack. The attacker first evicts target data from the Cache via eviction data. Then the victim executes its code. Finally the attacker accesses data from target data‘in the shared memory and measures if these come from Cache or system memory via timing.

If the system supports flushing Cache lines, usually through the *clflush* instruction in the ISA, then the attacker can utilise this to carry out efficient cross core attacks through shared LLC’s.

With the *clflush* instruction, the attacker is allowed to invalidate all entries from a specific memory address from the Cache hierarchy. Crypto libraries are often present on a system as a shared library, making it so any process can access lookup data used by the library, such as the T-tables for AES. Modern processors also implement page sharing or deduplication, allowing sharing of memory space for data that is accessed by multiple sources at the same time. This is done in order to prevent multiple contexts from loading multiple copies of the same data into the memory if its already present, allowing for performance benefits on shared systems.

In Flush+reload[12], the attacker flushes, lets the victim execute and then measure how long it takes to access these elements in the attacker process to see if they were put back into the Cache by the attacker. Flush+flush[13] starts in the same way, but instead of timing access to the shared memory, it times a second flush. The time this flush takes differs depending on if the data flushed is still present in the Caches or not.

Finally, specific hardware functionality can lead to even easier attack implementations. The Prime+Abort attack [37] uses the Intel TSX hardware to perform an even more efficient implementation of the Prime+Probe attack. The Intel TSX system is an implementation of memory transactions, where the results of an entire code section on the computer state can be instantly invalidated if the process is aborted. Aside from Prime+Abort being faster than Prime+Probe, the attack does not require any form of timing to retrieve its results and can wait for a hardware callback to conclude its measurement, instead of needing to wait for a process for an arbitrary time.

The ρ-VEX design currently has no shared Cache levels and no support for the flush operation. This limits our attacks to those that share a single L1 Cache (Prime+probe, Evict+time).

Timing-driven attacks

The attacker observes the timing of the cryptographic algorithm under attack, and derives a statistical model based on input plaintext and/or output ciphertext. These execution times correlate with the amount of Cache hits and misses occurring. Deriving information based on this execution time relies on creating models to derive theoretical collisions that can happen in execution: lookups that are done in the Cache vs lookups that come from memory.

A common method for timing attacks are Cache collision attacks. In these attacks, the attacker runs an encryption in an empty Cache. A Cache collision can happen when the same Cache line is accessed multiple times by the victim process. For these attacks to success, the probability for a iteration of the victim process where these collision do not happen need to significant enough so a

measurable difference can be found with a realistic sample size. A first example is the single encryption collision attack [36], single round collision attack. In the first round of AES, multiple lookups are made to the same T-table. Categorizing these samples on theoretical collisions results in a found minimum for the right categorization: the least amount of execution time due to the collisions. This attack can be also be performed on the last AES round, resulting in a more efficient lookup due to the use of a unique lookup table for the S-box operation used in this round for certain AES implementations [36].

Another category is Collisions happening between two encryptions. An example is the wide collision attack by Bogdanove et al. [8], where input plaintext pairs are chosen where a collision between two plaintext values is measured via five lookup collisions, a so called wide collision.

Other timing attacks are the evict+time attack, discussed in the previous subsection on access based attacks. A timing attack that relies on similar mechanisms is Bernsteins attack [7]. For Bernsteins attack, the attacker has access to the device under attack, or an exact copy of it. The attacker makes a template of the timing data based on known plaintext and known key inputs. Due to natural evictions happening during other system operations (for instance a server that handles incoming encryption requests via a certain protocol), certain plaintext input will have larger average execution times. The attacker can then use this template to correlate measurements on the device under attack using these stand out combinations, deriving the real key of the system under attack.

The ρ-VEX processor has a cycle accurate counter that can be accessed by the attacker to perform these timing attacks.

Trace-driven attacks

In trace driven attacks, the attacker gets (partial) access to traces of access patterns. These traces contain patterns of the sequence of Cache hits and misses. Acquisition of traces can be done via either power measurements to identify accesses to the system memory or via hardware based tracing functionality to track Cache performance. These often go via performance trackers, where a direct pattern of Cache hits and misses can be acquired.

The ρ-VEX processor has Cache performance trackers for each individual context, accessible to the user, making it currently vulnerable against these trace attacks. The performance trackers can be accessed by processes via the control registers.

3.2.2. Other Cache Attack Classifiers

1. **Cache behavior and interference:** the source of interference: whether these are Cache hits or misses, and whether these are caused by the attacker or within the victim process itself.

2. **Sharing level and degree of Concurrency:** how many processor resources and Cache levels the attacker and victim process get to share, and how much these can operate at the same time. For instance based on the system scheduler and multi-threading capabilities, and if the attacker needs to be active during the victim process or only before and after it. Because of the nature of the ρ-VEX processor, it becomes very easy to go from isolated Caches, to shared Caches by just remapping the hardware contexts to the different Caches with a reconfiguration. Because at this point of time of the development of the ρ-VEX processor, there is no formal rule on what level in operation can trigger the reconfiguration; only the operating system, that bases it on ideal performance, or also by the user processes based on requests. Multiple situations are thinkable:

    - No sharing: during operation it is made sure that the cryptographic algorithm never executes in the same lanes and thus never shares Caches with the process invoking it. The attacker can not request a reconfiguration by itself.
    - Sharing possible by operating system: The attacker can not request a reconfiguration by itself, but because the encryption is invoked by the attacker itself, the attacker process and cryptographic algorithm end up sharing a Cache.
    - No restrictions: The attacker is allowed to request reconfigurations at its own initiative.

3. **Based on the system types**: based on the type of processor, and if Virtual Machine operation is used. In our work single core attacks and multicore without shared Cache level are relevant to apply on the ρ-VEX platform.

4. **Based on the target crypto algorithm**: different algorithms have different weaknesses exploitable by Cache Attacks, and require their own method to derive information using knowledge on the implementation of the specific algorithm. In this thesis the focus is exclusively on AES-128. For instance an instruction Cache Attack on algorithms that have key dependent instruction flow, like possible on RSA [38], would not be applicable, as AES does not have key dependent instruction flow.
5. **Based on initial state of the Cache**: whether the attack requires a Cache clear, needs to create some initial Cache state or needs the victim process data in the Cache. Distinction between these is considered to not be relevant to choose what types of attacks to investigate. This distinction only tells how the attack should be implemented.
6. **Based on information of inputs**: Attacks on cryptographic algorithms can be divided in degrees of control and knowledge over the encrypted data: Chosen plaintext, where the attacker has the freedom to make encryption calls themselves. Known input/output, where the attacker only receives random samples of encryptions done by other processes. Finally attacks without knowledge on exact input/output are called asynchronous attacks. For these attacks, the attacker has to do statistical analysis of potential plaintext inputs, as they have no knowledge on the exact inputs. An example would be analysing Cache usage of encryptions on plaintext from the English language, demonstrated in [10]. In this thesis we will only consider synchronous attacks.

3.3. Detailed description of selected Cache Attacks

In this section, practical Cache Attacks considered to be used for this research are described. These are selected based on the fact that they are currently implementable on the ρ-VEX hardware (one private Cache level, no operating system, no attacks based on cross-VM mechanisms). The list of attacks considered, that are explained in detail in this section are:

1. **Prime+Probe Attack** [10] in Section 3.3.1
2. **Evict+Time Attack** [10] in Section 3.3.1
3. **Final Round Collision Attack** [36] in Section 3.3.2
4. **3rd round Wide Collision Attack** [8] in Section 3.3.3
5. **Bernsteins Attack** [7] in Section 3.3.4
6. **First round Trace based Attack** [35] in Section 3.3.5

3.3.1. The first round attack based on lookup probability scores

Attacks that have the attacker share the same Cache with the vulnerable code, almost always target the first encryption round because the most information is known to the attacker at that point. In an attack with lookup probability scores, the attacker targets specific Cache lines known to be used by the table data of AES. The attacker interferes with these Cache lines multiple times, and uses the acquired measurement data to derive the most likely accessed Cache line that is theoretically constant for their targeted input plaintexts.

The first round attack

The principal of the first round attack is to use the knowledge that the state used in the first round of lookups in the T-tables of AES, is simply a xor of the input plaintext bytes and the first roundkey bytes in Equation 3.1.

$$x_{i,j} = p_{i,j} \oplus k_{i,j} \tag{3.1}$$

The lookups in the first 9 rounds of AES-128, per column c_n are according to the following equation:

$$\begin{bmatrix} y_{(0,c_n)} & y_{(1,c_n)} & y_{(2,c_n)} & y_{(3,c_n)} \end{bmatrix} = T_0[x_{(0+c_n,0)}] \oplus T_1[x_{(1+c_n,1)}] \oplus T_2[x_{(2+c_n,2)}] \oplus T_3[x_{(3+c_n,3)}] \oplus K_{c_n} \tag{3.2}$$

This means that for these 9 rounds, with 4 columns each, there will be 36 lookups per 1024 byte T-table T_n. The lookups in the first round are directly related to the combination of the chosen plaintext

bytes and the key bytes. Also, 35 other lookups will be done to the same table during the full encryption.

Imagine a situation where we keep one single plaintext byte constant, and thus one single lookup in the first round constant, generate the other plaintext bytes randomly, and do measurements on what Cache lines are used by these encryptions. We call this result the measurement score Q, and we write it as Q_k(p, l, y). Where k is the targeted subkey, p the chosen plaintext byte at the same index at the target key, l the targeted T-table T_l and y is the targeted Cache line.
Every encryption, the same lookup will be occurring in 100% of the encryptions during the first round. The other 35 lookups are randomly distributed. For a Cache with Cacheline size δ in a 32-bit processor, the probability that this same Cache line is not accessed in the other 35 random accesses, is $(1 - \delta/256)^{35}$. With the common Cache line length δ = 16 this becomes a probability of approximately 0.104. In the case of the ρ-VEX, with Cache lines of a single 32 bit word, this probability is 0.872.
Given these probabilities, if an attacker acquires a feasible set of traces where measurements are done to determine if a Cache line was accessed or not, the most accessed Cache line and thus the highest cumulative Q_k(p, l, y) value will be the one targeted: the lookup that is based on the value of the xor between the plaintext byte and roundkey byte in Equation 3.1.
In situations where Cache lines aren't just a single processor word, the information from this attack can be limited to only a couple bits of the key bytes. For instance with 16 word lines, only the higher nibble (4 bits) of the key bytes can be found, as the different accesses within a line cannot be distinguished.

Measurement via Prime+Probe
The most simple implementation of an access based Cache Attack is the Prime+Probe attack. This because the access pattern used for this attack is possible on any hardware where Caches are shared between processes. In the Prime-Probe Cache Attack, an attacker first primes a large set of data in the Cache by accessing a random array of data from within its own process. The attacker waits for an encryption and then probes their previously accessed data, timing the individual lookups to decide if a miss or a hit was registered on these lookups, and thus concluding if the line was accessed or not by the encryption under attack.

The algorithm to acquire measurement score Q for combinations of targeted key bytes k, chosen plaintext byte p and targeted Cache line y is as followed.

1. Attacking a n-way Cache, where there are n Cache sets with m bytes of data each, the attacker allocates a single data array of size n x m.

2. The attacker carries out the ”prime” step; they access every element in their n x m data array to make sure they are all cached. In a Cache with multiple words per Cache line, one address per Cache line is sufficient.

3. The targeted AES encryption is done. Either the attacker directly knows when a single encryption is finished, or they try to wait for a certain time period to make sure the encryption finished.

4. The attacker carries out the ”probe” step; they access every element in their array again, and time how long each entry takes to access. The attacker sets a certain timing threshold to make a binary conclusion if the the lookup came from the Cache, or had to be reloaded from the memory.

Measurement via Evict+Time
The Evict+Time attack aims to evict a specific table entry while making sure all other table entries used for the encryption are in the Cache. The attacker then measures the encryption time, which should be higher if the eviction was done to a Cache line that was used for the encryption. Acquiring many of these measurements on multiple plaintexts will result in eventual outliers for the most frequently accessed plaintext and thus information on the key.

The Evict+Time method to perform the first round attack works according to the following method:

1. Attacking a n-way Cache, where there are n Cache sets with m bytes of data each, the attacker allocates a single data array of size n x m.

2. The attacker issues an encryption with a known plaintext. Making sure all table entries used for this specific encryption are in the Cache.

3. The attacker targets a specific Cache line, and with that a specific lookup address (or a set of addresses if the Cache line is larger) and evicts it by accessing n times at offset m in their allocated data.

4. The attacker issues an encryption with the same known plaintext as in step 4. The attacker times this encryption and this time becomes the measurement score.

Determining memory locations
In the descriptions above, the assumption was that the attacker knows the memory address of the lookups tables of AES. This however, might not be the case. Usually, the layout is known, and thus only the start address needs to be determined. The attacker can first profile on activity of Cache lines, if the system is relatively free of noise and thus only mainly the AES code will use the Cache. A different approach is to do the attack on the full Cache space, and then use the pattern of the most likely candidates for multiple plaintexts to determine the key value, instead of directly computing it.

The second round attack
The first round attack can not be sufficient enough to extract all keybits, or at least not enough keybits like in our example of Cache lines with 16 words; where the search space remains 4 bits per keybyte, or 64 bits total that results in $2^{64} = 1.84 \cdot 10^{19}$ key candidates still unknown. This search space is too large for brute force search. In order to extract the full key, the information retrieved in the first round attack can be applied for an attack on the lookups in the second round, in order to retrieve the left over keybits.

In the second round, we set up the equations that yields the input for the T-tables in the second round. We write this as a function of the input plaintext and input key, and write the transformation in a form that uses the S-Box transformation $S[x]$ step (so not the T-table notation). Let us write the equation for element $x_{2,0}$ at the start of the second round, after the AddRoundKey step.

$$x_{2,0}^{(1)} = S[p_{0,0} \oplus k_{0,0}] \oplus S[p_{1,1} \oplus k_{1,1}] \oplus 2 \bullet S[p_{2,2} \oplus k_{2,2}] \oplus 3 \bullet S[p_{3,3} \oplus k_{3,3}] \oplus S[k_{3,3}] \oplus k_2 \quad (3.3)$$

Similar equations can be constructed for the terms $x_{2,i}$ in the third row. Note that the term $S(k_{3,3}) \oplus k_2$ is the roundkey $k_{2,0}^{(1)}$, that is derived according to the key expansion algorithm in Section 2.3.3. Depending on the term, one or more key bytes are xor'ed at the end. The lower bits of these individual key bytes can be ignored, as they do not influence the Cache line accessed. The higher bits of these key bytes are known. For the key bytes that are used for a S-box, the upper bits are also known, but the last bits that are not known are now relevant for the Cache line accessed. There are 4 unknown lower parts of key bytes. In the common architecture, with 4 unknown bits per key, this means a search space of $2^{16} = 65536$ bits. If the most accessed T-table address for constant plaintexts $p_{2,i}$ is measured, then the most likely candidates for these lower bits of the key bytes can be computed.

3.3.2. Final Round Collision attack

The final round collision attack proposed in [36], is a implementation of an Cache Attack based on timing single encryptions in a assumed to be empty Cache. The attack relies on collisions between the lookups in a single encryption, where the execution time varies depending on if multiple lookups are done to table entries on the same Cache line. The AES implementation targeted is one where the rest of the rounds are done via the T-table implementation discussed in Section 2.3.4, but the final round uses a separate table to look up S-Box values directly.

At the end of the final round of AES-128, a final AddRoundKey step is done on the current encryption state x with roundkey k^{10}. The final lookup done and resulting ciphertext becomes that in Equation 3.4, where C_{shift} is the shift constant of the row the ciphertext i,j is in.

$$C_{i,j} = S[x_{i,j-C_{shift}}] \oplus k_{i,j}^{10} \quad (3.4)$$

This round consists of 16 lookups to the same 256 byte S-box. If these lookups are done in a Cache where all AES data is absent, achieved by the attacker forcing out all cached data or by waiting sufficiently long enough for other processes to claim the Cache, then the execution time will correlate with how many collisions happen in this round.

Now lets look at the situation where a collision between lookups happen. We target arbitrary output ciphertext bytes n and m. In the final round, these are computed using the final round roundkey (K_i^{10}) and the current AES state bytes $x_{0...15}$ as in Equations 3.5 and 3.6

$$C_n = S[x_i] \oplus K_n^{10} \tag{3.5}$$

$$C_m = S[x_j] \oplus K_m^{10} \tag{3.6}$$

In case, where there is a collisions between x_i and x_j, their values will be the same and will result to the same lookup:

$$collision => S[x_i] = S[x_j] \tag{3.7}$$

When this collision happens, we get the following equation of the xor between the targeted ciphertext bytes:

$$C_n \oplus C_m = S[x_i] \oplus K_n^{10} \oplus S[x_j] \oplus K_m^{10} = K_n^{10} \oplus K_m^{10} \tag{3.8}$$

Now, all samples in which the collision between these targets happens, will be grouped together under this same xor value. Even better, only the samples where this collision happens are able to be grouped under this sample:

Assume we have the right guess for which Equality 3.8 holds. In case of the collision, the xor of the s-box lookups results in 0: $S[x_i] \oplus S[x_j] = 0$ As every element in the s-box is unique, for every other combination of lookups the inequality holds that $S[x_i] \oplus S[x_j] \neq 0$, and will thus always result in a outcome that is not equal to $K_n^{10} \oplus K_m^{10}$

Then, because all other samples grouped under the other potential xor values are just randomly selected together, while the samples grouped under the correct guess of the xor of the targeted subkeys always share a collision, the specific ciphertext xor value that corresponds to the equality in Equation 3.8 and thus yields the xor of two key bytes will have on average a lower execution time than the other samples.

3.3.3. The 3rd round wide collision attack

The collision attack proposed by Bogdanov et al. [8] relies on collisions happening between two sequential encryptions, where the input plaintexts are chosen by the attacker. The plaintext used for the second encryption is based on the plaintext of the first encryption, to create situations where collisions appear between the lookups of the two encryptions.

The attack starts with the attacker making sure the AES data is not cached, through either doing Cache evictions to clean the Cache, or wait a sufficiently long time that other processes evicted all AES data. A first encryption is triggered. This encryption accesses some table entries and puts the AES data partially back into the Cache.

Then, a diagonal of the AES data is selected. The diagonals are shown in Figure 3.1. This diagonal targeted is a diagonal of the plaintext.

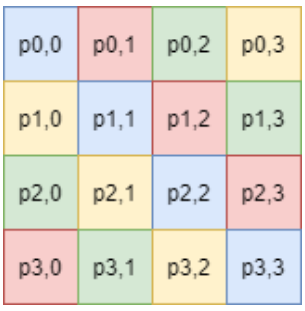


Figure 3.1: The diagonals of the input plaintext in the wide collision attack, each diagonal represented in a different color

The attacker first does their encryption on a random plaintext p_1. Using this first plaintext, the attacker generates plaintext p_2 by randomizing the bytes on the targeted diagonal, copies all other

bytes from p_1 and makes sure that $p_1 \neq p_2$. In Figure 3.2 the new diagonal is tracked through the first two AES rounds, and illustrates what happens when a wide collision between p_1 and p_2 occurs.

Assume during the second encryption, the diagonal of elements $x_{0,0}\ x_{1,1}\ x_{2,2}\ x_{3,3}$ is targeted. These elements are potentially different when compared to the first encryption. The other bytes that remain the same as the previous encryption are left blank.

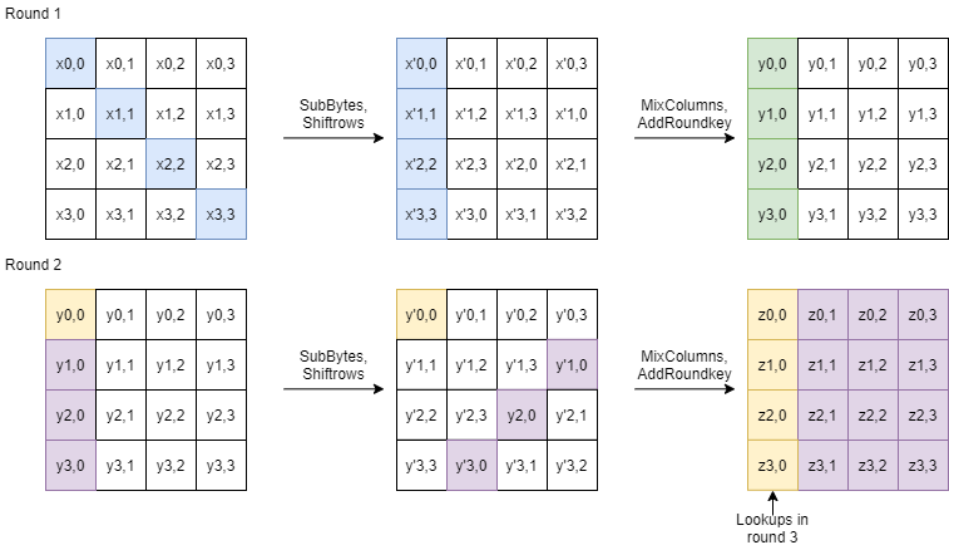


Figure 3.2: The wide collision being caused by a collision at the end of AES round 1. Tracking the blue diagonal through the first round. In the second round, a theoretical collision is highlighted in yellow. The purple values indicate bytes that may differ from the first encryption while blank values are guaranteed the same as in the first encryption

The steps of the first encryption round are carried out as seen in the figure, and after the Mixcolumn step, 3 columns ($c_1\ c_2\ c_3$) are identical to the first encryption, but column c_0 is different, highlighted in green. Now, because the the values in this column are all coming from the combined result of the four values in them before the MixColumn step, these values can be identical to that in the first encryption without having identical plaintext bytes used in the computation. This is unlike in the previous steps, since each byte was only dependent on a single input plaintext byte in the steps before.

In the situation where there is a collision; the same value as in the previous encryption is present in this round will result in a guaranteed Cache hit. A theoretical situation where byte $y_{0,0}$ of the state at the start of round 2 is identical to the previous encryption is highlighted with yellow in Figure 3.2. The other bytes in this column are assumed to not collide, and highlighted in purple to show that these are now different than in the first encryption, and we do not know anything about their value anymore.

Now, the collision in this round is also propagated to the next round. When we track $y_{0,0}$. The steps of AES are applied, and what can be seen is that, now the entire first column c_0 is identical to that in the previous encryption. The lookups done at the start of round 3 result in four additional guaranteed Cache hits as opposed to the situation where there was none. The other three columns are now all new values that we have no knowledge on.

Applying Wide Collisions in an Attack

To use the weakness of wide collisions the attacker constructs the following attack:

The attacker iterates over all 4 targeted diagonals. For every diagonal, n pairs of values for the diagonals (A,B) for $4 \bullet n$ targeted diagonal pairs total. For every diagonal pair, the attacker randomly generates I values for the other bytes, and constructs the plaintexts p_1 and p_2 with these, copying the same bytes to both plaintexts, but setting the targeted diagonal to the two values of A and B. Each of these plaintext pairs are sequentially encrypted in an empty Cache r times. The attacker thus does $4 \bullet n \bullet I \bullet r$ total encryptions.

For every diagonal pair (A,B) the binary decision is first made if a wide collision happened or not. The list of candidates for (A,B) are used in the final step. The algebraic equation of the collision needs to be solved based on these pairs. Equation 3.9 shows the expression of the targeted diagonal in

Figure 3.2. a_i and b_i are the elements of the first diagonal, a combination of 4 colliding diagonal pairs is required in order to solve this equality.

$$02 \bullet S[k_0 \oplus a_0] \oplus 03 \bullet S[k_5 \oplus a_1] \oplus 01 \bullet S[k_10 \oplus a_2] \oplus 01 \bullet S[k_15 \oplus a_3] = 02 \bullet S[k_0 \oplus b_0] \oplus 03 \bullet S[k_5 \oplus b_1] \oplus 01 \bullet S[k_10 \oplus b_2] \oplus 01 \bullet S[k_15 \oplus b_3] \quad (3.9)$$

Bogdanov et al. [8] solve this by iterating over all possible combinations of 4 diagonal pairs from all 4 + m proposed collision pairs (A,B) for a diagonal. Iterating over all possible key values to check if they satisfy the equality. They estimate the offline processing step to take $2^{32} \bullet (\binom{4+m}{4})^4$ AES encryptions. They experimentally found 66 million measurements, roughly 2^{26}, to be sufficient to get correct results.

3.3.4. Bernsteins attack

In Bernsteins attack [7] , the attacker has access to a device that is identical to the attacked device. The attacker performs a profiling stage on the copy of the device with known plaintexts and a known key, and tries to make a statistical model of the timing of encryptions based on these measurements.

The attacker targets a lookup in the first round, being $T_k[p_{i,j} \oplus k_{i,j}]$. Now, the timing of a full encryption can be correlated to the value of this lookup. This is mainly attributed to natural evictions happening in the process of calling the AES encryption. For instance because of operating system related Cache usage or because of interference with the memory location where the inputs are stored.

The attacker uses the example device to do measurements of every intermediate value for $n_{i,j} = p_{i,j} \oplus k_{i,j}$ for every targeted key byte. They calculate the average encryption time and the deviation through measurement. With the knowledge on the measurements, the attacker computes for every value of $n_{i,j}$ how the average of that value compares to the total average of the measurements for all values of $n_{i,j}$. These samples are ranked on most deviation from the average.

The attacker can do measurements on the attacked device, measure plaintext values of a targeted byte that averages the highest execution time: p_{max}. Then assume that this should be for which the value of $n_{i,j}$ was maximum in the profiling step: $n_{max(}i,j)$. The attacker computes $k_{i,j} = p_{max(i,j)} \oplus n_{max(i,j)}$

3.3.5. The first round collision attack based on traces

In a trace based attack, the attacker gains access to raw information on the hit and miss patterns in the processor during the encryption. These traces are formatted with H for hit, M for miss. H-H-M means that a processor does a sequence of three memory accesses. The first two are in the Cache and hit, the last one is missing and results in an access to the main memory.

The way an attacker can acquire these traces is done via two methods:

1. **Performance trackers**: If the CPU tracks their hit and miss rates via performance trackers, and the attacker has a method to read these trackers fast enough to registers hits and misses, then the attacker can acquire the Cache traces via these performance trackers.

2. **Power traces**: The difference between Cache accesses and memory access can be distinguishable in a power consumption trace from the processor, enabling the access behaviour trace to be retrieved this way [39]

A practical trace attack is to target the first round of AES. The attacker makes sure the AES data is fully evicted from the Cache by waiting long enough or doing evictions themselves. Then a encryption with known plaintext is done and the Cache trace is registered. The attacker tries to register a collision between the lookups after the first one in a specific table. A collision between lookups can happen with a probability of $\delta/256$, with δ the Cache line length in words. This is done in a adaptive manner: An arbitrary plaintext is chosen for p_i, then, the attacker iterates over all other plaintexts possible for the next plaintext p_j that shares the lookup tables. Once the attacker registers a hit on this second lookup, the attacker knows the following:

$$k_i \oplus p_i = k_j \oplus p_j \text{ for the upper 8 - } log_2(\delta) \text{ bits} \quad (3.10)$$

$$p_i \oplus p_j = k_i \oplus k_j \text{ for the upper 8 - } log_2(\delta) \text{ bits} \quad (3.11)$$

In case of an attack on the S-box implementation of AES, this attack can generate all inequalities for the 120 keybyte pairs.

If an exact pattern of hits and misses are registered, then colliding lookups can be identified.

The attacker can adaptively change their next input plaintext. If for instance, we find a pattern M-M-H-M in the lookups in table T_1 in the first round, then this collision could be between lookup 1 and 3, or lookup 2 and 3. Randomly permutating either the plaintext used in the first lookup, or in the second lookup will then give a clear answer on which two lookups collided. This will results in the xor values of key byte pairs $k_i \oplus k_j$

3.4. State of the Art of Cache Attack Countermeasures

As Cache Attacks have proven to be a serious security risk when executed on systems with protected data, the potential for Countermeasures has become a widely studied topic. Countermeasures have been successfully implemented and shown to be effective at either making Cache Attacks more difficult or fully preventing them. In this section we will briefly introduce Countermeasure concepts found in literature, and how they perform if implementations have been made.
We group the Countermeasures into the following categories:

- **Code Modifications:** Change the implementation of the AES algorithm to become more resistant or immune to Cache Attacks.
- **Reduce System Level Privileges:** Prevent certain steps required to execute an attack to be done by untrusted parties.
- **Cache Redesigns:** Alter Cache designs to remove the characteristics exploited by Cache Attacks.
- **Noise based Countermeasures:** Add noise to the measured Side-Channel to make attacks require more samples.
- **Attack Detection:** Detect Cache Attacks in real time, and act to prevent them when detected.

For Cache Attacks targeting AES, Countermeasures that can partially operate during execution can be implemented to only function during the vulnerable rounds: for most attacks the first or final encryption rounds, and in specific cases the second round (using data from the first round), or the second round plus third round for the Wide Collision Attack in Section 3.3.3, these Countermeasures are only required to be functional during those rounds.

3.4.1. Code Modifications

Avoid Cache or memory

A first solution is to avoid memory lookups all together, so no information on data dependant lookups can ever leak into the execution of the algorithm. In case of the AES encryption algorithm, the S-box lookups can be replaced by the computation done to compute the lookup data. This however has the downside of slowing down encryption time significantly Another method is to completely bypass all system memory by using dedicated AES hardware on the system, for instance secure crypto coprocessors or specific hardware support for cryptographic functions, such as the AES instruction set on Intel processors [23]. These AES specific hardware resources not only avoid Cache leakage, but also provide significant speedup to AES throughput, with Intel reporting an average of 1.3 cycles/byte [40].

Bitslicing [21] is another promising implementation of AES that bypasses the use of tables and thus data dependent lookups all together. It computes multiple encryption in parallel. For this implementation, N input plaintexts are taken together and encrypted in parallel, where N is the bit length of the processor words used (32/64 bits). The input plaintexts are first shuffled in such a way, that 128 processor words each have 1 bit of every plaintext, in sequential order. The S-box transformation is implemented via its mathematical definition. The other rounds can be implemented via simple operations on the full words, with shifting and xor additions on full words. A more efficient implementation of the bitslicing method is proposed in [22], where only four plaintexts are mixed into each other to have more efficient throughput. This saves a lot of memory operations as all data for one encryption call can be stored in the registers of the processor as opposed to the implementation in [21]. This faster

implementation is found to only be 5% slower than the reference implementation of AES in a setup on an AMD Opteron 146.

A different solution is to bypass the Cache and access data in memory only. How the adversary can do this depends on what functionality the current platform supports. If the adversary can allow processes to bypass the Cache by temporarily disabling caching, then Cache usage is completely hidden. This however, comes at a large performance cost as every lookup has to come from memory directly, what can lead to a slowdown of a factor up to 100 [10], depending on the memory architecture and AES implementation.

Secure table implementations

The vulnerability to Cache Attacks for AES specifically comes from the use of lookup tables. Although the most frequently used 32-bit implementation uses four 1024-byte T-tables, other implementations can be used at cost of performance. A first transformation done is to avoid the use of a separate S-Box in the final round, and instead use transformations of the T-tables to get the S-Box lookup values. This covers the significant weakness of the final round using an isolated table with less lookups than the other tables. This implementation currently is the standard in crypto libraries like OpenSSL.

Multiple implementations that try to manipulate AES lookup tables so they become more resistant to Cache Attacks have been proposed [41]. The first method is to only use T-table T0, and cyclically shifts the bits in these to derive the values for T-tables T1 to T3, according to the definition in Section 2.3.4.

Another method is to use the classical implementation of AES, using the S-Box is the only lookups and computing the other steps. Due to the smaller size and more frequent accesses (160 lookups), analytical Cache Attacks can become unfeasible on systems with larger Cache lines. This implementation however is measured to be 3 times as slow as the T-table implementation [41].

If this implementation still proves to be vulnerable to Cache Attacks (like tracing), then the S-Box can be implemented as individual lookups per bit [41]. These are grouped as individual, bitwise s-boxes that fit on a single Cache line. Lookups will thus always result in the same Cache line being used, at the cost of overhead for the extra lookups and the shifting of bits to get the final result. The calculated performance cost to do this implementation for a protected round is around 60% extra cycles when compared to the regular T-table implementation.

Cache state normalization

In order to distort attacks relying on direct Cache accesses from the attacker in the same Cache, the adversary can make sure that every time the AES code runs, all Cache lines containing AES data get cached by adding in extra code that makes accesses in order to load all table data in the Cache, accessing a set of table entries based on Cache line length. This practice is referred to as Cache warming in [42]. A second method is to, temporarily prevent data in the Cache from being evicted. This requires specific hardware support to enable such a mode. An example is called ”Cache as Ram mode” in [43]. The adversary will first load all AES data into the Cache, and then lock in all data until it is considered done with encryptions.

3.4.2. Reduce System Level Privileges

Prevent Cache sharing

The vulnerability from access based attacks comes from the fact that a spy process gets to share a Cache with the victim process. It is possible to prevent this sharing, managed by the operating system. An approach would be to lock the Cache to the encryption process, and before another process gets access to the resources, the used Cache is flushed [10]. This disables the information leakage through the Cache between two processes, but comes at a high performance price and doesn't fully prevent all Cache related attacks, such as those that rely on timing a single encryption [7] [36]

Prevent shared libraries and shared memory pages

Cross core attacks like flush+reload [12] rely on shared software libraries and memory deduplication, where multiple processors end up sharing the same data from memory and thus the same items on shared Cache levels. Preventing access to these secure libraries by untrusted parties can prevent these attacks. Disabling Memory deduplication also disables this attack, but will hurt memory efficiency.

Obfuscate system timers and Performance Counters
Since the timing of a single AES encryption requires access to precise timers, the system timing abilities can be either blocked to non trusted users, making them rely on imprecise external timers, or they can be made less precise be setting the minimal unit of time to a higher amount than clock cycles. An example is masking the value returned by a clock counter register to only return the upper i bits [44]. To prevent attackers from performing a trace attack, the Cache Performance Counters can be blocked from user access, or time to access can be limited so they can't be precisely synchronized with the execution.

3.4.3. Cache Redesigns

Cache partitioning
A way to prevent Cache sharing is through the use of Cache coloring or partitioning: a mechanism implemented in some systems to divide Cache into groups to determine what processes can share parts of the Cache[45]. Although Cache coloring is designed for increased Cache performance, it can also be utilised as a Countermeasure against Access Based Cache Attacks [46]. Hardware Cache partitioning [17] is also a effective Countermeasure against Cache Attacks. The Partition Locked Cache (PLCache) method in [17] also implements locking of Cache lines, where internal Cache evictions and thus Side-Channel leakage because of these internal evictions are prevented on protected lines.

Randomized Cache mappings
In order to relate Cache accesses to data accessed, the attacker needs to have precise information of the mappings of the (virtual) memory addresses to the Cache lines. These mappings can be obfuscated. The authors in [17] propose the Random Permutation Cache (RPCache), where the mappings between memory addresses and Cache lines is randomly permutated per process during runtime. An attacker can thus measure what Cache lines are accessed by the victim process, but does not know what addresses are loaded on these Cache lines and can thus make no useful conclusions on victim data accesses. This new Cache design was measured to have only 0.3% performance decrease for the entire system. Another similar Cache redesign called NewCache is presented in [18]. In this design, performance benefits of direct mapped Caches are layered on a dynamic memory-to-Cache mapping. Aside from its security benefits, this design also improves power consumption and reduces the miss rate of lookups.

3.4.4. Noise based Countermeasures

Timing noise
Cache Attacks that rely on acquiring timing traces of AES encryptions can be made more difficult by trying to obscure the timing information. Making the timing behaviour more noisy by introducing some random stalls or dummy operations during the AES encryptions, and making the attack require more samples and less practical in the process. Including this noise can also be done in the system that implements the system timer. Adding noise to the returned time value of the system timer, instead of manipulating the actual measured value can also obscure Cache Attacks that require precise timing information.

Access noise
Noise can also be generated on an access level. Additional code running on the device during the encryption that does random accesses to the memory associated with the encryption algorithm, or evict random Cachelines can make an attack more difficult. This creates noise for the attacker, which can not distinguish what source accessed these Cache lines. This is either implemented via extra steps in the AES encryption function, or via extra threads in the processor, like the flush+prefetch Countermeasure introduced in [47] that aims to obscure Cache usage in shared Cache levels. They show their method to actually increase performance of the victim process by 10.2% in their best case setup, as prefetching improves the encryption throughput, and measure only 22.3% of leakage compared to 96.7% in a single decryption.

3.4.5. Attack Detection

For access based attacks, detection mechanisms have been developed that aim to detect memory access patterns that can belong to these access attacks, and deploy a certain Countermeasure to

interrupt these attacks once an attack is detected. Examples are NIGHTs-WATCH [19], which analyzes file executables before running them , SCADET [20], which keeps track of performance tracker during runtime to detect attacks, and LiD-CAT [48], which has its own hardware component to monitor Cache access patterns to detect attacks.

3.5. The ρ-VEX Countermeasure potential

In Section 3.4 we discussed Countermeasure designs proposed in literature. In section, we apply the theory we discovered while surveying the potential for the ρ-VEX architecture to implement Countermeasures with its reconfiguration system.

3.5.1. ρ-VEX reconfiguration patterns

Let us first define all distinctive reconfiguration scenarios for context0, refer to Section 2.2.3 for an explanation on the reconfiguration words used:

1. The current context gets more execution lanes and more Cache space: i.e. 0x0011 to 0x0000
2. The current context loses execution lanes and Cache space : i.e. 0x0000 to 0x0011
3. The current context moves to another set of instruction lanes and Cache: i.e. 0x0011 to 0x1100
4. The current context gets more execution lanes and Cache space, but in different Caches: i.e. 0x2210 to 0x0011
5. The current context loses execution lanes and Cache space, but in a different Cache: i.e. 0x0011 to 0x2210
6. The current context is disabled: i.e. 0x0011 to 0x8811 or 0x2211

The Reconfiguration types 1. and 2. are unique to the ρ-VEX processor architecture. The execution time can be varied by assigning different amount of instruction lanes and Cache space to a hardware context.

Reconfigurations of type 3. effectively a context switch between two active contexts, swapping their resources. It is a unique feature in the ρ-VEX to do this directly into hardware. A context switch in a regular CPU is the process of setting up a new software context in the hardware context. It is usually implemented by storing the current state of the context (the register file) to a memory stack, and loading in the new register contents related to this new executing software context. Where a context swap in a regular CPU has measured overhead via this system could be in order of 10,000 cycles [49], making it unfeasible on regular hardware to briefly switch contexts from processor resources. The process of frequent context swaps is detrimental to the performance to these traditional CPU's. On the ρ-VEX however, the reconfiguration allows processes to swap execution sources in 10's of cycles, making it realistic performance wise to do this often and thus a realistic approach to implement schemes against Cache Attacks.

Reconfiguration 4. and 5. are combinations of the former. Reconfiguration 6. is to disable a context from executing, to free up the resources to other hardware contexts.

3.5.2. Countermeasure: Cause timing noise through reconfiguration

Lane size configuration

Whenever reconfigurations are done, the following effects influence the execution time of a program:

- How many instruction lanes are assigned to the program and the ILP of the program.
- The amount of Cache hits/misses introduced by the reconfiguration
- The amount of overhead cycles from the reconfiguration request itself

A brief experiment was done on the ILP of the AES algorithm compiled by the ρ-VEX compiler. This includes the used T-table implementation as well as the slower S-box implementation described in [33]. The code is executed on the ρ-VEX simulator. This simulator does not simulate the Cache, and has its

simulated memory delay set to 1 cycle per lookup, so only the ILP matters for the execution time. The results can be found in Table 3.1.

	2-way	4-way	8-way
S_Box : Full encryption	2771	2242	2051
S_Box : round 1 AddRoundkey + SBOX	157	141	139
T_Table: Full encryption	625	415	347
T_Table: Round 1 (= round 2/3/4..)	67	43	34
T_Table: Final round	54	35	28

Table 3.1: Measured cycle counts for each configuration. No simulated Cache, MEMread = MEMwrite = 1 cycle

As a Countermeasure, varying the amount of execution lanes assigned to the victim process can thus make timing based attacks more difficult. When configuring to larger configurations, only the ILP becomes a deciding factor, and as shown with the experimental results in Table 3.1 there is enough ILP to have a different execution time for all configuration sizes. This also means that there is the option to cause noise by speeding up the throughput of the victim at the cost of other resources sharing the processor.

When reconfigured to a smaller configuration, the Cache state becomes relevant, and Cache misses can become a cause of additional noise. This is caused by cached data not being properly being shared between the larger and smaller configuration, both in the data and instruction Caches.

Timing noise through shared memory resources
Like discussed in Section 2.2.4, parallel processes can interfere with each other on the memory bus. If a transaction with the main memory is done by a context, and another context needs access to the main memory as well, then it is stalled till the previous transaction is complete. In case of a memory write, Cache and memory access of other contexts are also stalled, till the write is finished and the Cache consistency is guaranteed. If the ρ-VEX is shared between two running contexts, then they will interfere with each other and cause timing noise in measurements.

3.5.3. Countermeasure: Cause access noise within a single processor

Since the ρ-VEX allows for frequent and relatively fast hardware context switches between Caches, reconfiguration patterns are implementable that obscure Cache usage by moving contexts over the Caches.

An approach could be to briefly swap two active contexts from execution lanes during vulnerable parts of the victim process, and thus briefly swap the Caches. Because Caches can be briefly switched, accesses can be prevented from being done in a certain Cache, and additional Cache evictions from a different processes can briefly execute in that context, resulting in disturbance in the Side-Channels.

3.5.4. Countermeasure: Cause access noise on shared Cache levels

The ρ-VEX architecture could also be used to implement schemes that dynamically execute different contexts that try to obscure the state of higher level shared Caches, like with the flush+prefetch Countermeasure in [47]. The fast switching between active contexts could be a very efficient way to implement this Countermeasure on a ρ-VEX core. However, in this thesis we limit ourselves to a standalone ρ-VEX core, and the implementation of such a scheme is left for future work. This implementation would also require significant updates to the ρ-VEX hardware to make it work on a system with multiple processor cores and shared Caches.

3.5.5. Countermeasure: Prevent Cache collisions within a vulnerable algorithm

Vulnerable parts of the victim process can be spread out over a maximum of four isolated Caches in an 8-way ρ-VEX. This way, collisions that happen between the lookups of an encryption in a cleared Cache can be prevented from happening and attacks that exploit them can thus be made more difficult or even fully prevented, depending on how many collision opportunities there are left. The T-table implementation for instance conveniently has 4 lookups to the same table in a round, making it possible to split up all these lookups over individual Caches. Performance penalties for this method can however be detrimental, as the victim process has to reconfigure four times in quick succession, can

lose instruction lanes in case reconfiguration to a smaller configuration is required and has to reload instruction data to the Cache for every reconfiguration, in the current architecture.

3.5.6. Countermeasure: Prevent Cache sharing between processes

Because the ρ-VEX can quickly reconfigure to change what Cache is currently assigned to the running context. We consider two practical applications:

- Secure schemes of operation can be implemented, where a (untrusted) process that does a call to the encryption function never shares the Cache with the executing function.
- Detection schemes that detect an attack could use a reconfiguration to prevent the attack from succeeding, by moving the process out of the attacked Cache. This has theoretical performance benefits as instead of drastic measurements that are usually taken on attack detection, like full Cache flushes or interrupting the system.

3.6. Selecting attacks for experimentation

In Section 3.3, some Cache Attacks from literature were described to give examples on how varying Side-Channels are used, how the attacks are exactly implemented and how much time they take. The attacks were also selected because they can be implemented on the ρ-VEX hardware in its current implementation.

In this section, we will briefly go over these attacks again, and describe if we consider these attacks practical to test. We also consider if these would specifically be vulnerable to a Countermeasure concepts presented in the previous Section. Finally, we decide what Cache Attacks to implement and test the Countermeasures against in the rest of this thesis.

Prime-Probe attack

As we discussed in Section 3.5, we do have the potential Countermeasure ideas that logically hide lookups into other Caches during execution. This means that this will specifically have an effect on access based Cache Attacks, and we want to see how much we can counter these attacks with our Countermeasure designs. The question is if we can make this uncertainty statistically significant to make access based attacks require more samples, while keeping acceptable overhead. We want to at least have a access based attack for this, and we identify that of the ones discussed in Section 3.2.1, only the Prime+Probe attack currently has a practical implementation on the ρ-VEX hardware with only one shared Cache level.

Single encryption collision attack

Single round collision attacks are valuable to experiment with, as the Countermeasure of splitting up rounds to prevent Cache collisions can uniquely protect against this attack type. As we know that it will fully prevent attacks on rounds with only four colliding lookups, an implementation of the final round collision attack with 16 S-box lookups can thus be more interesting subject to this Countermeasure. As some information leaking would still be left. The collision attack on this implementation of AES (final round implemented with S-box) should also be more efficient than one where the T-tables are substituted, allowing for more experiments. The offline phase of this collision attack rather simple when compared to for instance the wide collision attack, and thus repeating this attack multiple times is feasible. The attack also already yields clear intermediate results of information on its measurements, making it possible to analyze even without the final processing steps, but based on the correctness of the data going into the final step. It is also possible to test a trace based attack on the same collisions, with the embedded Cache performance counters in the ρ-VEX. However, due to the time constraint of project it was decided to focus on the other attacks first, and trace attacks end up not being included in the experiments for this thesis.

Evict-Time attack

The Evict+Time attack is a rather unique attack, since it combines the principals of an access-based attack (targeting certain Cache results), but the measurement Side-Channel is not a direct access based but timing of an encryption. Experimenting with this attack also allows us to see the effect of the Countermeasures on a setup process, as the attack requires an initial setup encryption.

Wide collision attack
The Wide collision attack by Bogdanov et al. is implementable on the current ρ-VEX hardware. However a combination of this being a attack with high sample count required 2^{26} and lengthy offline processing phase that can take hours or days [8], makes it rather unfeasible to repeat many times in feasible time. A strong difference between the final result resolution (the amount of retrieved keybits can vary from run to run) also makes this too impractical to perform multiple times. This attack however can still potentially pose a threat if Countermeasures are implemented to be round specific, and are not active during the second and third round. The ρ-VEX also has the potential to allow for unique Countermeasure influence on this attack, as it can actively tamper with the setup process that the first encryption of the diagonal pairs of this attack does.

Bernstein's attack
We identify Bernstein's attack as an interesting research opportunity, as we can disturb both the model created as well as the live tracing data acquired. It becomes especially interesting on the ρ-VEX, as the statistical data of the template can be influenced by the workload on the other ρ-VEX resources, and this might be difficult to properly emulate for an attacker that is making their template.

We however decide against testing Bernstein's attack for the following reasons:

- Bernstein's attack can quite lengthy, requiring in the order of 2^{20} to 2^{24} samples for practical setups [50] when compared to the $2^{13.3}$ samples for Evict+Time [10] or 2^{15} to 2^{18} samples for the final round collision attack in [36] , on top of needing to do both template creation and live attack for every setup, making it less practical to do a lot of variation testing of Countermeasures that have variable settings that are testable.

- Bernstein's attack requires natural evictions to happen during execution, coming from either OS overhead or specific memory accesses done when encryption are processed by the device. Since we do not have a given specific setup for the ρ-VEX, and are testing it as a bare minimum processor, we do not have the given setup where we would test the attack on. Emulating it is possible, but might give a unrealistic model of how the attack would work in practice, more than with the other attacks discussed.

Summary
We the select Prime+Probe, Evict+Time and Final Round Collision Attacks to investigate our Countermeasure designs on. We argued against the Bernstein and Wide Collision attacks, because they are deemed impractical because of large sample size requirements or complicated processing steps. We see trace based collision attacks as a interesting attack, but we keep these attack out of the scope of this thesis because of time limitations of the project.

3.7. Conclusion

In this chapter, we first described how the Cache of a processor can pose a vulnerable Side-Channel. A broad overview of Cache Attacks against AES in literature is given, alongside classifiers commonly used in literature. We identified a list of attacks that are implementable on the ρ-VEX in its current design. These attacks are the Evict+Time Attack, Prime+Probe Attack, Final Round Collision Attack, 3rd round Wide Collision Attack, Bernsteins Attack and First Round Trace based Attack. These attacks are described in detail to understand how they work and how a practical implementation would look like. We selected the Evict+Time, Prime+Probe and Final Round Collision attacks to test in this thesis. The other attacks were deemed to take too many samples to efficiently test multiple times, and trace based attacks were left out of the scope of this thesis. An overview of Countermeasures found in literature was given. These Countermeasures were divide into the categories Code modifications, Reducing System Level Privileges, Cache Redesigns, Noise based Countermeasures and Attack Detection. Then, the Runtime Reconfiguration system of the ρ-VEX was analyzed, identifying the possibility to influence execution time through lane sizes assigned to a process, and the possibility to influence access behaviour by moving contexts between different caches. We proposed five Countermeasure concepts: Causing timing noise by changing configuration sizes, generate access noise within a single shared ρ-VEX processor, efficiently generate access noise in higher Cache levels shared with other processors, prevent internal Cache collisions within a vulnerable algorithm and finally implement efficient systems

that prevent Cache sharing between processes. We additionally identified the shared memory system between contexts to be a potential source of measurement noise.

4

Cache Attacks and Countermeasures implementation on the ρ-VEX

This chapter describes the implementation of the studied Cache Attacks on the ρ-VEX, and the design and implementation of three Countermeasures based on the ideas from the previous Chapter. In Section 4.1 practical details regarding the ρ-VEX architecture that are relevant for the attack and Countermeasure implementations are discussed. In Section 4.2 the implementations of the three Cache Attacks, Prime+Probe, Evict+Time and Final Round Collision are described. Then, the design and implementation of the three Countermeasure designs are described. Section 4.3 describes the Countermeasure where configuration size variations are done to cause timing noise in the attack measurements, called *n-Lane*. Section 4.4 introduces *CacheSwap*, where executing hardware contexts briefly swap between Caches. Finally Section 4.5 describes *ScatterRound*, where vulnerable rounds of the AES algorithm are split out over multiple Caches to prevent internal Cache Collisions.

4.1. Practical details for Attack and Countermeasure implementation on the ρ-VEX

This section introduces some practical details when working with the ρ-VEX to implement Cache Attacks and the Countermeasures against it. In Section 4.1.1 we give a detailed explanation of how access to the reconfiguration system and certain system information can assist an attacker, and what assumptions we make for our attacks in regards of these privileges. Section 4.1.2 describes practical details of the ρ-VEX that are relevant when implementing attacks. Section 4.1.3 describes the process of how to determine a new configuration word for Countermeasure purpose, and how we deal with that in our implementations. Finally, Section 4.1.4 describes how we will implement Countermeasure related reconfigurations in the AES code.

4.1.1. Assumed Privileges in the ρ-VEX

In this thesis, we do not use an operating system on the ρ-VEX. We directly compile and run code together with a small standard library for the ρ-VEX. Aside from missing operating system overhead, this also means that we do not have formal rules on how much control the attacker has over the reconfiguration system. In case of limitless control, the attacker could just force the the system in a certain configuration and that would leave our Countermeasures in this chapter that utilise reconfigurations useless. The following assumptions on system privileges are made:

1. Configuration during encryption During encryption, the attacker cannot reconfigure anymore. The control is given to this encryption process. Attacks where the attacker operates from a parallel context to the attacked encrypting context, and can thus issue reconfigurations during the encryption, or attacks where the attacker can utilise the debug bus to reconfigure, are left out of the scope of this research. Attacker and the attacked algorithm are thus executed from within the same context. This emulates a setup where the attacker makes its encryption calls from a shared library such as openSSL [51].

2. Knowledge on system configuration The attacker always has a way to figure out what configuration size it is executing in: if the attacker has system privileges to access the control registers, then the current configuration word can be accessed. If there is no access, then multiple tricks can be thought of to still get at least the current issue width. Such as a small prime+probe attack on the Cache, so see if it a 1-way, 2-way or 4-way Cache that is currently connected. Or by measuring the time of some section of the code with at least some instruction bundles with a length higher than 4. These measurement take time, and thus detecting brief reconfigurations can be unreliable or impossible without access to the control registers. Hiding the configuration size is thus not considered as a potential Countermeasure. We will also show that our attacks can be made to work regardless of the configuration size in Section 4.1.2.

3. Attack phase reconfiguration A significant detail that divides the attack scenarios is if the attacker can reconfigure during the attack phase or not. It depends on if a practical operating system permits non-trusted sources from reconfiguring. To test the Countermeasures, we will first consider the scenario in which the attacker has no access to reconfiguration, and see if the attacker can improve from there if they gain access of reconfiguration in their attack code. Like discussed in point 1, we assume the attacker is never allowed to reconfigure during the AES function call itself. Access to reconfiguration includes access to disable other contexts. Since we consider the situations with or without other contexts operating as a separate test case , we are indirectly testing the effect of the attacker turning those off by comparing these results with an isolated context attack.

4.1.2. General Attack implementation details

This section describes the practical considerations that have to be made in order to successfully implement the Cache Attacks selected in Section 3.6. This includes considerations made based on the unique Cache architecture of the ρ-VEX.

Dealing with memory addresses

In this thesis, the process of determining the memory location of the AES tables will not be part of the attack process. We will create a setup where we know exactly where the tables are allocated and our attacks will work with that in mind. The reason for this approach, is that first it makes a successful attack require less samples and thus less time to execute many times in profiling stages, and secondly it will make interference from the processing of the acquired samples on the processor in the attack preventable by mapping that data to Cache lines that are not used for AES.

Dealing with the configurable Cache

The configurable Cache of the ρ-VEX is described in Section 2.2.4. Due to the reconfigurability, the Cache line selected to put the data of a certain memory address is different based on the configuration size. This is because the Cache acts as a directly mapped Cache for its current size. This has two practical implications:

1. A certain address can get mapped to different Cache lines based on the configuration. For instance, in a 2-way configuration using lanegroup0, everything maps to Cache0. Once the configuration size increases, the same data can now mapped to both Cache0 and Cache1, based on its address. This is illustrated with an example of the different mappings between the configurations 0x0888(2-way) and 0x0088(4-way) in Figure 4.1.

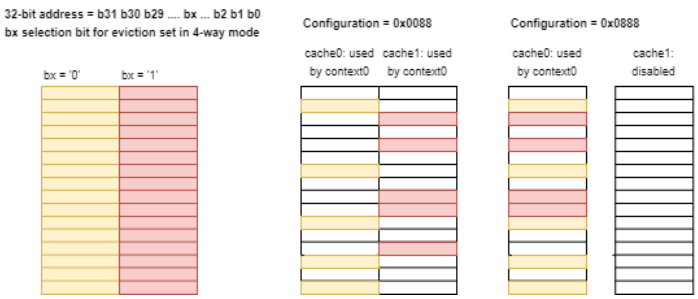


Figure 4.1: The different mappings of the same addresses in different configurations

2. A data word can be stored on a Cache line where it does not map to according to the eviction scheme for the current configuration. This is caused by the fact that it did map to that line in a previous configuration. This is illustrated in Figure 4.2. First, an eviction is tried of a specific memory address. If we started in the current configurations 0x0088 this succeeds. If our context executed in a 2-way configuration before this (0x0888), this data will be on a different Cache line, and will stay there until that line specifically is evicted. On the next load, it will appear on the line where we expect it.

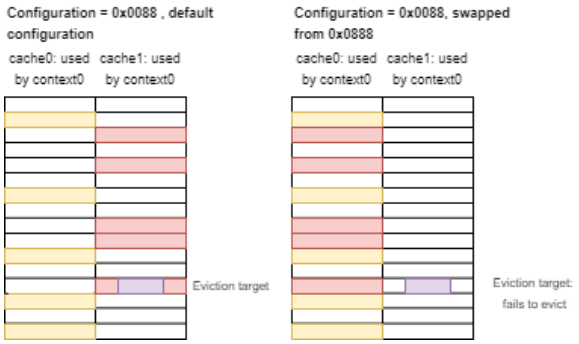


Figure 4.2: How evictions can fail under certain reconfiguration patterns

Because detection of the current configuration requires some overhead, and because we want to show that it is not necessary to know the configuration to carry out these attacks, we implement our attacks to work under any of the configuration sizes: We adapt our Cache access schemes according to point 1., by always mapping the addresses under the assumption of the maximum Cache size, 4 Caches. If two memory addresses map to the same Cache line in the largest configuration, then they will also map to the same line in smaller configurations. We adapt to the second fact, that in case we want to evict some specific data out of the Cache, we will evict 4 Cache lines, the Cache size separated from each other. In the largest configuration, we make sure all Cache lines that the data could be in at that moment are evicted. In smaller configurations, this will simply result in some extra evictions of our own data that already evicted the data we wanted to target.

Data acquisition
For data extraction on the ρ-VEX, we are limited to using a USB UART connection to access the memory on the board. The ρ-VEX debug bus allows for memory dumps over a given address range.The measured data transfer speeds using the debugging software, over a 11.52 kB/s serial UART connection are 9.5kB/s upload speed and 9.0kB/s download speed. Datasizes larger than 32 Mb (transfer speed 1 hour) can take impractically long to transfer. Because of this limited transfer speed, we will do processing of the data on the ρ-VEX as much as possible, and then communicate the results. In order to make this practical, we will implement our attacks with tracing checkpoints: only on trace counts defined by the user the intermediate results will be reported.

4.1.3. Determining a new configuration word

For Countermeasures that require to reconfigure the system, a practical obstacle is that the new configuration words need to be determined based on our Countermeasure criteria, but also based on the current configuration of the system. There are a couple requirements for our system that determines the new configurations associated with our Countermeasures:

1. We need to make sure that the previous configuration is stored, so the optimal configuration determined by the processor is restored after a Countermeasure briefly reconfigures.

2. The system that handles the current configuration must be implemented in such a way that these random configurations done by a executed process is handled properly. If for instance the AES process claims all processor resources, the system must be aware that it still wants to execute the other contexts and eventually configure back.

3. If the system configuration system decides to issue its own reconfiguration, this new request should be respected when the AES code attempts to switch back. The ability to ”lock” a config-

uration could be desirable, so that another source can not reconfigure during the brief reconfigurations related to the Countermeasures.

4. Before reconfiguration, the current configuration must be checked to determine the new configuration. We do not want to interrupt other contexts if not necessary. For instance, if we execute our AES in context0, are in configuration 0x0012 and decide that we briefly want to assign less lanes to context0, we need to correctly reconfigure to 0x0812 instead of a configuration like 0x0888.

5. We need to make sure that the system always executes in a configuration that allows our desired reconfigurations. For instance, if we execute AES in context0 according to the following configuration 0x0011 and want to configure to less lanes for AES for the n-Lane Countermeasure, then 0x0811 is illegal according to rule 3 of configurations in 2.2.3, and we cannot do our desired configuration pattern from this configuration. We would need to reconfigure to 0x1108, so it would be more ideal if the default configuration in this scenario was 0x1100.

We decide that, for purpose of this research, we bypass all these practical issues in our experiments, and make it so our setups can always properly do configurations in the patterns we want. The practical details on how to implement the process that determines the correct configuration is left as an open question, and is greatly dependent on how the processor is practically used and what the operating system allows.

In the setups where the processor is shared with another context, we set up the base configuration that allows for all reconfigurations required for the Countermeasures to be issued. No contexts will be dynamically started or stopped, so these reconfiguration words can be determined at the start. This makes is that the Countermeasure has full control and passively tracks the current configuration. Reconfiguration back will always be to the same default configuration defined at the start, and a reconfiguration from our default configuration will be guaranteed legal on the system.

4.1.4. Random reconfigurations in code

In order to implement Countermeasures that reconfigure during the AES code, a general method is required to efficiently implement these random reconfigurations in code.

In the implementations of Countermeasures in the rest of this chapter, we will implement the majority of the random reconfigurations in the following way:

1. Before the call to the AES code and potential timer start for the attack, generate all random reconfiguration words required for the AES function call.

2. Store these words in a small lookup table, where the relative Cache index of this table does not interfere with the measurements.

3. Whenever a random configuration at a certain moment is required in the AES code, implement this as a lookup to the predetermined table, and write it to CR_CRR.

We decide to implement this because of the following reasons:
1) We consider the possibility of the process of presenting a random reconfiguration to the process based on a specific setting made by the process (a new control register to write to), the current configuration and some hardware implementing a (pseudo) random number generator a possibility. This would then possibly be implemented via another register that reads as this new configuration value, and will thus yield a comparable implementation and AES performance.
2) Implementing the processes that determines the reconfiguration during the AES encryption call itself in constant time might be difficult because of data dependant control flow (both instruction count and instruction Cache interactions), and together with the extra cycles become a significant disturbance to the measurements. Although this is beneficial to protecting our data, this is not beneficial to our objective to purely measure the influence of executing in different configurations. We know that adding some randomized stalls interfere with the effectiveness of the attacks, but we want to purely see what the configuration pattern does, especially since we identify the possibility to determine valid reconfiguration words in hardware instead of code. This setup will more accurately tell the influence of the reconfiguration independent of what implementation might turn out to be most ideal
3) An added benefit of this implementation of reconfiguration via writing to CR_CRR is that if the current configuration is written, the reconfiguration process is not started. These writes thus do not need to be

conditional to enable us to decide to reconfigure or not, but the current configuration can just be written in those cases and nothing will happen to the configuration.

4.2. Cache Attack implementation

In this section, we will describe our implementations of the Evict+Time, Prime+Probe and Final Round Collision attacks.

4.2.1. Evict+Time Attack implementation

We implement the Evict+Time attack based on the description given in Section 3.3.1.

Online phase

For the online part, we make the following decisions in implementing our attack:

- The target plaintext byte values have been limited to a subset of 9 values. Although the attack could theoretically be implemented with only one target plaintext value in this situation, profiling the attack resulted in the requirement of multiple plaintext values because the attack itself was interfering with the measurements of 4 of the 16 plaintext bytes. We decided to go with 9 plaintexts, the first '0000 0000' and the following 8 each with only one of the bits set to '1'. We emulate the attack as if we do not know about the memory location, and derive our key based on the pattern in the measurements when setting the different target key bits. This also adds the benefit of adding more measurements in a single attack, painting a better picture of the influence of Countermeasures that rely on randomness.
- As we know where the tables are located in memory, we only directly target the table entries related to the targeted subkey with evictions.
- We include the option to clear the Cache before the attack. We do this because if this process runs in isolation, the setup encryption before the eviction only reloads the elements targeted by the eviction. Clearing the Cache emulates that the setup process actually does an important step in the attack, as that might be necessary if the practical processor this attack is executed on has a lot of other processes using the Cache. This might be relevant for the effectiveness of certain Countermeasures.

A pseudo code implementation of the evict- time attack can be seen in Algorithm 3.

Algorithm 3 Evict+Time attack

```
1:  for TargetKeyByte = 0, 1, ... 15 do
2:      for TargetBitSet = 0, 1, ..., 8 do
3:          for TraceIteration = 0, 1 ... N do
4:              Set random plaintext bytes
5:              Target byte in plaintext gets set to '0"s, TargetBitSet becomes '1'
6:              for EvictionAddress = 0, 1 ... 255 do
7:                  optional (partial) Cache clear
8:                  encrypt with generated plaintext
9:                  evict target n addresses for current Cache size
10:                 encrypt with generate plaintext and time
11:                 store measurement sample
12:             end for
13:         end for
14:         if TraceIteration in TraceCheckPoints then
15:             copy current results to final result array
16:         end if
17:     end for
18: end for
```

The data acquired consists of an int array of dimensions:

evict_data [**N_TraceCheckPoints**] [**TargetSubkey** = 16] [**TargetKeyBit** = 9] [**EvictionSet** = 256]

The eviction sets in these traces are aligned with the targeted table data location in the memory for every subkey, so the eviction sets directly map to the table indexes used for the attack on the subkey.

Offline phase
The data from the attacks is loaded into MatLab. For every eviction set, we subtract the average encryption speed of that eviction set from all samples targeting that set, like proposed in [10], to compensate for evictions that hit other cached data like the roundkeys or plaintext.

Solving directly with individual samples: Figure 4.3 visualises the measurement results of one specific keybyte target. A darker colored sample means a higher average AES encryption time and thus a higher probability the key candidate is the correct one.

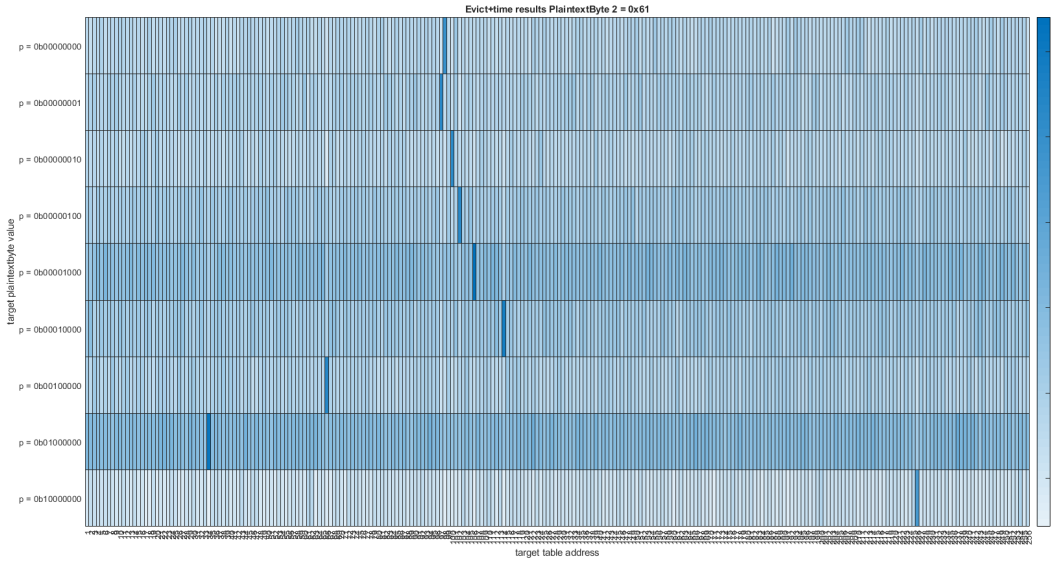


Figure 4.3: Evict+time, visualisation of measurement data on plaintext byte 2

The measurements for each plaintext value are clear enough to individually show the key value. We solve: found lookup $T[n]$, with n = k $\oplus$ p.

- p1 = 0b00000000, result n = 97 : k = 0x61
- p2 = 0b00000001, result n = 96 : k = 0x61
- p3 = 0b00000010, result n = 99 : k = 0x61
- p4 = 0b00000100, result n = 101 : k = 0x61
- p5 = 0b00001000, result n = 105 : k = 0x61
- p6 = 0b00010000, result n = 113 : k = 0x61
- p7 = 0b00100000, result n = 65 : k = 0x61
- p8 = 0b01000000, result n = 33 : k = 0x61
- p9 = 0b10000000, result n = 225 : k = 0x61

A potential way of solving now, is to do this calculation for all 9 plaintext targets of the results and take the key candidate that appears the most often as the most likely candidate.

Solving with the all samples In the case of not knowing the memory location exactly, these results

would be part of a larger sample size: eviction would be done to every Cache line instead of the lines where we know the table data is present. In that case, the pattern in these most likely candidate reveals the key byte:
Comparing the location of the most likely accessed memory address compared to p = 0b00000000 shows the value of the individual key bit at that index: if the keybit is 1, we xor with 1 the resulting bit becomes 0, which means the value gets reduced and the measured maximum will be to the left. The opposite for keybit = 0 happens, then maximum index moves to the right as the output value increases. We see the pattern of the most the likely Cache index, from top to bottom:
decrease - increase - increase - increase - increase - decrease - decrease - increase.
Converting this to the observed key bit pattern discussed:
1 - 0 - 0 - 0 - 0 - 1 - 1 - 0
While keeping the correct bit order in mind we get the value 0b01100001 or 0x61, our known key byte. We implement these processing steps in a MatLab script. We add the following features:

Verifying the key MatLab processes the samples from the input file, and estimates the key based on both the procedure described above, where the key is either derived from the pattern of the most likely candidates, or by selecting the majority measured key value. MatLab reports the key and how many key bytes are correct for the known key.

Showing the correct samples If we get a result where not every key was retrievable, we would like to know how this is caused. If only one of the 9 subsolutions used for this attack was incorrect, the entire key will be incorrect if derived from the pattern of maximum values. We call these correct subsolutions instead of correct keybits because the results can still reveal the full key, with all bits correct. Going by these values represents better how effective Countermeasures are than just only looking at the correct key guesses, since we identified multiple methods to do this with our data.

Computing the keyrank For every subsolution, we compute how the correct eviction set ranks among all samples. We look at the plaintext were setting every individual plaintext byte to '1' discussed previously. These are 8 measurements for each of the 16 targeted keybytes. This means that we will check how many of 128 "subsolutions" have the correct result.

Computing the AES performance Because the Evict+Time execution conditions (relative stable Cache state) gives a good estimation on how the AES encryption speed would be under regular execution (without our attack), we also report the average encryption time for this attack, to determine how much our Countermeasures influence the throughput of AES encryptions.

4.2.2. Prime+Probe Attack implementation

Online phase

The general procedure during the online phase of the Prime+Probe attack matches with that of the Evict+Time; we generate the same measurement combinations: targeted keybyte, plaintext byte values out of the 9 values like described in 4.2.1 and now targeted Cache set via the prime+probe method.

Algorithm 4 Prime - Probe attack

```
for TargetKeyByte = 0, 1, ... 15 do
    for TargetBitSet = 0, 1, ..., 8 do
        for TraceIteration = 0, 1 ... N do
            Set random plaintext bytes
            Target byte in plaintext gets set to '0"s, TargetBitSet becomes '1'
            Prime all Cache in current configuration
            Encrypt generate plaintext
            Probe all Cache, storing the measurement scores
        end for
        if TraceIteration in TraceCheckPoints then
            copy current results to final result array
        end if
    end for
end for
```

Measuring individual memory accesses

During tests with the implementation of the Prime+Probe attack, we found that the ρ-VEX compiler didn't directly allow us to reliably time a single memory lookup. The compiler reschedules the instruction where we access CR_CNT freely, resulting in failing to measure our access times.

This reordering can be prevented by either rewriting the compiled code at assembly level, or by changing the compiler optimization. We were no able to find direct support for pragmas in the ρ-VEX compiler that would allow to annotate these specific lines to prevent them from being rearranged. Because we want to keep our code close to a realistic implementation of AES-128 on the ρ-VEX we decide not to change the optimization level. Because we are compiling often because our variations with the Countermeasure are for a large part directly embedded in the code (to prevent overheads that would come with making them runtime configurable with parameters to disturb these measurements), manually changing the assembly code was also deemed impractical because of the many times we compile new code.

Ultimately, it was decided to do the probe step, that measures a single memory access time, in a function call. This correctly forces the order of operations, but introduces a new problem in that a single timing sample will have different execution times based on the configuration size, due to function call overhead code having different execution times in different configurations. We even measured that a Cache miss in the largest configuration is the same amount of cycles as a Cache hit in the smallest configuration. This can be resolved by setting a specific threshold based on the configuration, or by doing that attack on the total time of accesses instead of binary hit/miss measurements using a threshold. We decide to go for this last solution, as it should be the easiest to implement, and is also naturally more resistant to noise from the shared memory interface.

Offline phase

The offline part mirrors the offline step of the Evict+Time attack in Section 4.2.1. We do not subtract the average of every probed set as this is not required for the binary accessed/not accessed measurement.

4.2.3. Final Round Collision Attack implementation

The final round collision attack is implemented as described in Section 3.3.2. We target the implementation where the final round is implemented with via 16 S-box lookups (referred to as using table T_4).

Online phase
During the online phase, we clear the Cache, generate a random plaintext, time an encryption with this plaintext and group this measurement based on the output ciphertext.

Algorithm 5 Final round collision attack

for $Trace = 0, 1, ... N$ **do**
    Clear the target Cache
    Optionally clear other system Caches
    Generate a random plaintext
    Encrypt the plaintext and time it
    From the output ciphertext, compute the xor terms of all 120 byte combinations
    Store the result for every byte combination at the right xor value
end for
Compute the averages of all byte combination values

Offline phase
We implement the following pseudocode algorithm in MatLab. First, the final round key is computed via the AES code used in this project and filled into MatLab. From these key bytes, the correct values for the Xors between the key bytes are calculated.

To ease the analysis of the attack, we will stop at the step where all Keycandidates of the final round key are generated in our experiments. The amount of correct ”subsolutions” is reported instead. When there is a majority of correct subsolutions, the key should become retrievable.

Algorithm 6 Final round collision attack

for $CipherByteCombo = 0, 1, ... 119$ **do**
    Find the index of the lowest average execution time: MinIndex
    Convert MinIndex to bit representation: MinIndexToBits
    CipherXorGuess[CipherByteCombo] = MinIndexToBits
end for
for $TargetBit = 0, 1, ... 7$ **do**
    for BitStringValue = 0,1,…65535 **do**
        Convert BitStringValue to a bit string of 16 bits
        Check all CipherXorGuesses, count how many are satisfied for the current bit candidates
    end for
    Report bit string that satisfies most Cipher xor guesses: SolutionCandidateString[TargetBit]
end for
generate all 256 KeyCandidates from all combinations of SolutionCandidateString[$0 ... 7$] and xorBits(SolutionCandidateString[$0 ... 7$]), by doing key inversion.
for KeyCandidate = 0,1 …255 **do**
    verify if the KeyCandidate is correct with plaintext - ciphertext pair.
end for

4.3. n-Lane: Noise via random configuration size variations

This section describes the design and implementation of a Countermeasure that utilises random variations of the configuration size of the ρ-VEX. We call this Countermeasure n-Lane.

4.3.1. n-Lane Design

The first proposed Countermeasure is to utilise the ability of the ρ-VEX to assign a different amount of execution lanes to a specific context. When assigning different amounts of lanes to a context, the context gains the ability to execute more or less instructions per clock cycle, as far as the ILP of the current program allows. The amount of measured cycles can thus be altered by changing this configuration during runtime.

As a Countermeasure, the adversary can try to randomly alter the amount of execution lanes assigned to a context. In the meantime, lanes becoming available can be assigned to other contexts, or

other contexts can potentially be stalled if the AES context requests all available lanes. We call this Countermeasure n-Lane.

Two scenarios are possible for this Countermeasure. The first is where there is only execution time influence caused by the ILP, the instruction throughput that varies for different configuration sizes. In this setting, the Cache data should be perfectly shared between the configurations, and the implementation of AES should thus run in such a way to only one Cache' address space is used for the data. The second scenario is where we do not keep the potential Cache disturbance in mind. This means that for a reconfiguration from a larger configuration to a smaller one, the encryption process itself can disturb the Cache state from its regular configuration, by allowing evictions between its own data that would not happen in the default configuration. This can disturb measurement samples beyond the encryption that is ran in a different configuration size.

If we can both reconfigure to a larger and a smaller configuration (so the baseline configuration is a 4-way configuration), then there is a theoretical balance possible where we do not reduce our encryption throughput, at the cost of stalling other contexts that share the processor.

The following subsections highlight the practical details that are relevant for the implementation of the Countermeasure:

Losing data between configurations

As we vary the amount of instruction lanes between the allowed configurations: 2 lanes, 4 lanes or 8 lanes, we can lose or gain access to different Caches when compared to the previous configuration. If we make reconfigurations where the cached AES data is lost, significant performance decreases will result. So in order to implement this Countermeasure, and only get performance variations from the ILP of the AES code, one Cache needs to be included in all these configurations. The 2-way and 4-way configurations are then selected in such a way that they have a shared Cache unit in their configurations. The 8-way configuration naturally has access to all Caches.

Placing the AES data

An additional concern, even if we keep a constant Cache between all configurations, is that the default mode of operation can place the data in the Cache in such a way that parts of it or the entire range of data is lost when reconfiguring to a smaller configuration. This is illustrated in Figure 4.4, where the placement in the Cache is no issue when going from 4-way to 8-way or 2-way to 4-way, but can become a problem from 4-way to 2-way or 8-way to 4-way.

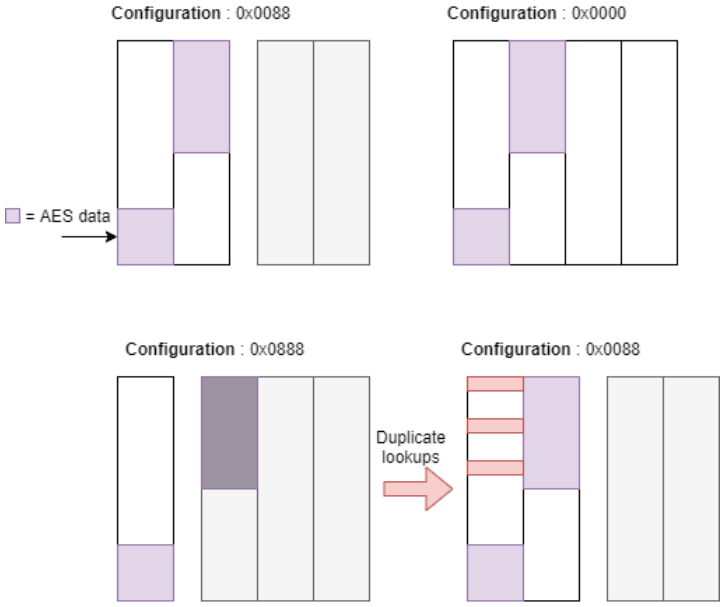


Figure 4.4: Effect on Cache when running the n-Lane Countermeasure on an encryption that is executed in 4-way by default

Against this effect, the adversary implementing this Countermeasure has a couple options to minimize the effect:

1. Make sure that the interference of other processes is minimal. If under normal operation, the processor gets to do many encryptions in the same core undisturbed, then the perceived performance hindrance is relatively low. The encryptions will just eventually cause two copies of certain table data to reside in the Cache at the same time.

2. Influence the address of the AES data. Although it is likely not easy to do this with instruction addresses, it is possible with the data by allocating a sufficiently large array and place the table data based on the start address of this table. This way it perfectly forced in the Cache belonging to only one lanegroup.

3. Only configure to larger configurations in this Countermeasure implementation.

4. Don't keep this property in mind, and see if the slowdown of the Countermeasure is still reasonable.

Comparison with random delays
The direct contender to the n-Lane method, is to insert random delays into the AES function itself. An implementation would be to do a random amount of dummy operations or execute some random amount of NOP instructions at the end or start of an AES encryption.

The implementation on the ρ-VEX via variations with the execution lanes has a few upsides compared to inserting random delays:

- Access to speedup as a way to add measurement noise, as opposed to only slow down. This at the cost of interfering with other processes, that might have a lower priority as opposed to encryption. This allows us to effectively move the cost of this Countermeasure to other lower priority processes instead of slowing down the effective throughput of AES encryptions. Think of a system that has to encrypt and decrypt messages over a communication channel under certain deadlines.

- The pattern might naturally occur in the processor under certain use cases. Think about a situation where the processor often decides to change lanes assigned to contexts to fit the workload. This means that under normal operation the ρ-VEX could already add measurement noise to time based Cache attacks. To amplify this, a lower additional percentage of random reconfigurations could be added to amplify this noise.

- As discussed before, the reconfiguration has the potential to load data in the Cache on different lines than with the default configuration, potentially making access based attacks like Prime+Probe more difficult as well.

Why we consider the n-Lane Countermeasure as a very realistic option to achieve disturbance in timing measurements, is because although the raw throughput of the encryptions is slowed down, the resources of the processor are not wasted in the meantime if other contexts are executing. like adding in timing noise through random stalls would. While the AES encryption forces a reconfiguration to smaller configurations, the other lanes can be used by other contexts running in the processor, potentially speeding them up.

In a situation where we get the proper setup to prevent many Cache misses coming from this method, and only doing a small fraction of the encryptions in another configuration, then the relative slowdown should stay within reasonable ranges.

4.3.2. n-Lane Implementation

For n-Lane, we want to reconfigure to different issue widths to gain execution time variations through the varied use of the victim process' ILP, and study the influence of potential Cache disturbance that happens because of the reconfiguration pattern.

We identify multiple potential ways to implement the general concept of this Countermeasure.

1. **In software:** At the start of every encryption, configuring back at the end.

2. **In software:** Between every round of the encryption, allowing for more potential variation in the noise.

3. **In software:** For multiple sequential encryptions.

4. **In hardware:** At any point during the encryption

For the experiments in Chapter 5 we will mainly implement the n-Lane Countermeasure as a reconfiguration at the start of the AES encryption, and by default configure back to the default configuration after the function has executed. This also gives us the option to exclude the entire reconfiguration process from the measurement data in case of timing attacks. We do this in order to only measure interference from the configuration changes, and prevent the code that decides the configuration, with potential variable execution time, to be included in these measurements.

We acknowledge that this will result in a very easy workaround for an attacker, that is, to cluster the measurement results and drop a percentage of the highest and lowest execution times. However, we also do a small experiment with reconfiguration between rounds, in order to compare this to the full encryption reconfiguration interval. These results should be indicative of the potential of the Countermeasure in implementations where the reconfiguration happen at different intervals.

We set the chance to reconfigure as a configurable parameter. We allow to either reconfigure to only large configurations, only smaller configurations or to both. The default is to do both. The percentages of which either occur can also be altered. This defaults is a 50/50 chance to select either of the two configurations associated with the n-Lane reconfiguration. The default n-Lane implementation in a 4-way execution mode, with a 10% chance to reconfigure, has a 5% chance to configure to a 8-way mode for a single encryption, a 5% chance to configure to 2-way and a 90% chance to do nothing and execute a AES encryption in the default configuration.

4.4. CacheSwap: Access noise via lanegroup swaps of two contexts

This section describes the design and implementation of a Countermeasure that utilises brief random swaps of processes between Caches to generate access noise. We call this Countermeasure CacheSwap.

4.4.1. CacheSwap Design

A Countermeasure that has interesting potential when another context is executing in parallel to the encrypting context, is to randomly swap the contexts between resources and then switch them back after a brief time. We call this Countermeasure CacheSwap. We require the operation to be in either a 2-way or 4-way mode reconfiguration to fully implement this idea, as we require the ρ-VEX to operate as multiple parallel processors. We ideally have at least one other context executing in the processor at the same time.

We identify the following effects this pattern has on the attacks:

1. The context that is switched to the default AES execution resources, does some Cache evictions, partially destroying the assumed Cache state on following steps of the encryption.

2. During the time that AES context has access to a different context, execution based on the assumed Cache state before the round is obscured and the part of the encryption acts like it didn't happen in the main Cache, obscuring information leaking from further lookups and potentially removing leakage from targeted lookups.

3. the reconfiguration overhead and the Cache misses resulting from this process (both data and instruction) are a noise source in timing measurements used for timing based attacks.

A random Cache swap can also potentially disturb an attack process that is executing a setup step or processing step of the attack. Think of the Prime+Probe attack that can be disturbed in either Prime or Probe step of the attack by a brief Cache swap. However, in practice this would mean that any non trusted process should have frequent swaps in their code, which would result in unacceptable performance costs that influence all processes on the processor. Thus we will only look at the potential to do swaps during the victim process.

4.4.2. Theoretical effect on specific attacks

Prime+Probe

This method prevents a small amount of the targeted accesses to be made in the currently targeted Cache. For an individually targeted lookup, this is equal to the percentage selected for the random Cache swap. The context that is briefly swapped into the main Cache can do random accesses that can add additional noise to the measurement. Since this attack directly measures if a Cache set has been accessed or not, moving this to another Cache will remove the information directly and invalidate the trace sample. For this to be significant, one would expect higher probabilities to be required. There is a potential security risk in this Countermeasure however. If the attacker has the option to freely configure to also access the second Cache, and if they can somehow guarantee a workload with a low amount of Cache accesses (or at least lower than AES), then the attacker can decide to attack the second Cache instead, potentially getting an even more efficient attack based on the CacheSwap reconfigurability process and workload in the second context.

Evict+Time

The CacheSwap Countermeasure can result in the setup process of the Evict+Time attack to not be successful, as the setup process uses a single encryption to guarantee a certain Cache state, resulting in a incorrect measurement sample. The attacker could work around this by doing the same encryption multiple times, to lower the probability of the Cache state being incorrectly initialised, or by manually loading all AES data in the Cache if that is something the attacker can do. The same chance happens in the measured encryption after the setup step, where the attacked access is done in a different Cache and thus invalidates the trace.

The CacheSwap also causes timing noise, by introducing extra random Cache misses in an encryption. A reconfiguration adds random noise by itself, adds extra Cache misses in the part that is executed in the different configuration and can cause random Cache evictions of the AES data add extra misses after the reconfiguration.

Final Round Collision attack

Randomly swapping to a different Cache generally leads to certain collisions expected to happen being prevented from happening, both by the setup lookups of the encryption being redirected to the other Cache, or by the expected second lookup that collides with the earlier lookup being redirected. This also means that for higher percentages, the performance of the attack is actually expected to increase again since then the majority of lookups will share the same Cache again, only now in the other Cache. An additional benefit is, that if the attack relies on the attacker doing Cache clearing before the sample themselves, they might be prevented from clearing the other Cache. This makes it possible that certain AES data is still present from a previous swap and yield an invalid measurement sample.

Other attacks

From the other kinds of Cache Attacks discussed in Section 3.3, we briefly analyze the theoretical effects:

Against *Trace Attacks*, collisions can be caused to be absent in a the trace while they are supposed to happen, invalidating a set amount of traces. This will likely only become significant to the effectiveness of the attack for higher swap probabilities.

Against *Bernsteins Attack*, effects similar to those in the Evict+Time attack are possibly found, as it relies on similar principals as that attack. Where instead of evicting many addresses on the same plaintext, we test many plaintexts on some arbitrary evictions that the system under attack does.

Against *the Wide collision attack*, which attack relies on 5 collisions split over round 2 and 3 between two encryptions. This means that this would require the CacheSwap Countermeasure to also be active during those rounds. Also, theoretically the entire third round should execute in the other Cache at once, since only swapping single columns would still leave the other collisions in the measurement data, having an diminished effect on the effectiveness of the Countermeasure. The Countermeasure would thus need to be implemented for this attack specifically by implementing it during the third round, and likely swap for the full round. This should then be more efficient for lower swap chances, as higher chances above 50% would lead to the collisions happening more often again.

4.4.3. CacheSwap Implementation

We implement the CacheSwap Countermeasure on the Cache attacks running in a 4-way ρ-VEX context. Between every column of the targeted AES round. The first round for Prime+Probe and Evict+time, the final for Final Round Collision. These reconfigurations are stored to not disturb the Cache state and submitted like described in Section 4.1.4

Before the encryption, a small function determines what AES column should be in what Cache, based on the percentage chance to swap Caches. This small process is not included in the measurements, as its implementation is not a constant time functions.

This implementation potentially causes multiple swaps in a single AES encryption round. For instance swapping for the first column, swapping back, then swapping again for the third column and swapping back again. After the final column is computed, the Countermeasure always switches back to the default configuration for the rest of the encryption.

4.5. ScatterRound: Preventing internal collisions via spreading operations over multiple Caches

This section describes the design and implementation of a Countermeasure that spreads out a round of the AES algorithm over multiple isolated Caches to prevent internal Cache collisions. We call this Countermeasure ScatterRound.

4.5.1. ScatterRound Design

A more specific Countermeasure against attacks that rely on collisions in the lookups in the encryptions, is to spread the lookups of targeted rounds over multiple Caches so the expected collisions do not happen at all. This is similar to the CacheSwap method, but opposed to the random Cache swaps we will now guarantee specific rounds to be split over the available Caches. This method also does not rely on other processes to do random Cache evictions in the main AES Cache, but could be added to the implementation to generate even more noise to the encryptions. We call this Countermeasure ScatterRound.

We do this split based on columns, as our state is computed column wise, and we can thus easily split the four lines of C code over the Caches. Figure 4.5 illustrates what happens with the lookups in a round that is split up over four Caches via this method.

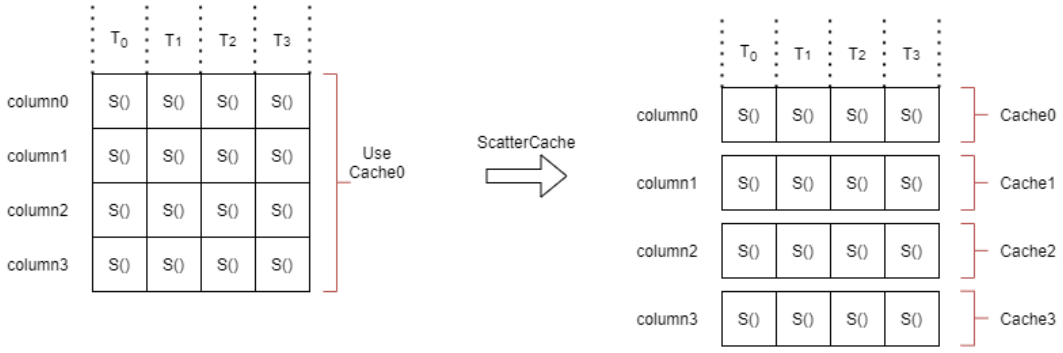


Figure 4.5: Visualization of splitting up a AES round over four Caches with the ScatterRound Countermeasure, indices T_x indicate what T-Table is used in case the round is implemented with the use of T-Tables.

Instead of collisions being possible between all 16 lookups, now only collisions can happen between the lookups done in the same column. In case of one single lookup table, we thus lose the information of collisions that happen between the elements in different columns. In case of a T-Table implemented round, collisions can't happen at all, since no two lookups to the same table are done in the same Cache in this round.

Practical implementation scenarios

There are multiple scenarios in which this technique could be applied. If forced on a configuration that has a default 2-way size, then the overhead costs are of the four reconfigurations required, and the additional Cache misses caused by execution in the other Caches. This can be quite costly, especially

since multiple bundles of instructions need to be loaded in every context. If proven to be effective, then maybe the design of the ρ-VEX could be altered to decouple the mappings of the instruction and data Caches from the execution lanes, such that only the data Caches can be reassigned while keeping the current instruction Caches mapped the same.

If applied in larger configurations, then there is relatively more slowdown, as the split up round requires extra cycles when compared to the default configuration, because the instructions executed in this round have less instruction lanes than with the default configuration given to the AES encrypting context. This configuration scheme can also cause the internal Cache state of the main configuration Cache to be disturbed, both data and instruction, leading to additional slowdown from the ideal execution.

Another potential approach is to set up a special execution mode, where four individual contexts do four encryptions in parallel, and swap system sources in their vulnerable round. This would cost up to three context switches per hardware contexts per encryption, but adds the benefit of a lot of the associated data staying in the Caches between encryptions. Operating system overhead would be required to execute a certain mode, as all other contexts need to do a software context switch to start encrypting. If a lot of encryptions are done after each other, the average overhead will eventually approach just the reconfiguration cost, as this context switch is only a one time overhead at the start of the first encryptions. The synchronization between the contexts is also an issue. If a context is carrying out its splitting up of a protected round, then ideally the other three encrypting contexts should not interfere with this. Synchronization is required, and contexts that want to do the split up round need to be stalled if another context is currently doing a split round.

Comparison with existing Countermeasures

On any CPU architecture, effectively the same effect could be achieved by making 3 copies of tables accessed in the targeted round, and computing every column with a lookup to a unique table. This however has some downsides:

- The Cache space occupied by the targeted tables increases with a factor 4. If this is done against the T-table implementation, then the required space for all these tables increases from $2^{12} = 4kB$ to $2^{14} = 16kB$, which can cause significant slowdown by making it so not all these entries can fit in the Cache.
- The memory demand increase could also be of significance if the implemented architecture does not have a sufficiently large main memory that would a memory size increase of that order (12kB in case the 4 T-tables are stored 3 extra times).
- The use of more tables means the lookups done during AES are more isolated, making them an easier target of access based attacks.
- ScatterRound potentially has extra benefits against attacks that do not rely on Cache Collisions. Further explained in Section 4.5.2.

When implemented on an isolated context that does encryptions and allowing the ScatterRound Countermeasure to spread one round over 4 Caches, it logically behaves similar to disabling the Cache on the final three column lookups of the round. However, there are some upsides when compared to Cache disabling:

- The ρ-VEX currently cannot disable data Cache usage for a single context, neither can it directly request a single lookup from memory instead of Cache. Making it impossible to implement this without also hurting the performance of other contexts.
- Cache disabling on small parts of code might be difficult to implement, especially if the operating system is designed without it in mind. If the operating system is not implemented with this in mind, the Cache could be kept disabled when software context switches happen, leading to large performance costs.
- This method can theoretically have more data hits on some of the lookups, if the other workloads do not have a high amount of memory accesses, while the Cache bypass method will surely need to access memory on all of them. This means theoretical performance increases.

However, there are downsides as well:

- On the current ρ-VEX hardware, instruction misses will also be caused by this Countermeasure which add extra performance costs.
- Evictions to another processes cached data are done and they are also briefly stalled, causing performance decreases for workloads sharing the system.

4.5.2. Theoretical effect on specific attacks

We analyze the theoretical influence of scattering the Cache lookups on the following attacks:

Final Round collision

In the unprotected case of our final collision attack, we end up with a relation between all 16 key bytes. We iterate over one targeted keybyte from 0 to 255, and compute the other 15 keybytes based on this decision. We iterate over all possible options on a known sample to verify which is the correct key value.

If the Cache lookups are spread out over 2 or 4 Caches, then only collisions can happen is between subsets of the keys. Instead of using all 120 inequalities, we can only use the inequalities belonging to two sets of 30 inequalities, or four sets of 6 inequalities. In the baseline attack, we knew the relations between all keys and only had to check all variations of one targeted key value, to compute all values of the other keys. This meant that the final verification step was to do $2^8 = 256$ encryptions. If split over two Caches, we have two isolated systems of inequalities, and have to match for all combinations of the two, and thus needs to do $2^{8*2} = 65536$ encryptions. When split out over 4 Cache, this becomes $2^{8*4} \approx 4.3 * 10^9$ encryptions of offline overhead.

These larger key candidate spaces do complicate the attacks, but the search space of $4.3 * 10^9$ key candidates is still within feasible brute force search with a known plaintext - ciphertext combination. The attacker also to adjust their attack with the Countermeasure in mind, as they cannot use the invalid subkey combinations in determining the most likely bit values.

If the Countermeasure is combined with a implementation where the final round is implemented without S-box lookups, but with shifts of T-table lookups, then the attack should be fully prevented, as no lookups of the targeted round can collide.

Additionally, if we had access to a theoretical 16-way ρ-VEX, with 8 execution lanegroups and thus 8 Caches, then we could potentially alter the code of the AES implementation to be able to split over these 8 Caches. This yields a search space of $(2^8)^8 = 1.8 * 10^{19}$ which would in fact make it impractical for brute force search, and nullifies this type of attack. However, such an implementation would likely be too impractical.

Evict-time attack

Doing every single column lookup spread out over the same Caches will not directly remove information that the attacker tries to retrieve with the Evict+Time attack. However, this Countermeasure can significantly complicate the attack:

- A single encryption before the eviction no longer guarantees that the table data is properly loaded like it needs to be for this attack to work. If the Countermeasure makes it so the second encryption does a column in another Cache than in the first encryption, the acquired sample does not contain the expected information.
- The attacker has to keep in mind that the Cache line to evict might be positioned in another context Cache than in the main targeted context, complicating the attack, or even leaving it invalid if the attacker can not configure to that Cache by itself.
- If another context runs in the processor, it can interfere with the attack, as it has the chance to evict lookups that are done in Caches allocated to other contexts in the main configuration.

Prime Probe

The effect of the ScatterRound Countermeasure against the Prime+Probe attack depends on the current system privileges of the attacker, and can either result in making the attack more difficult but also potentially easier.

When our attacker is naive and only keeps scanning for the Cache address where the attacker expect the T-table lookups to map to, then only 1/4th of the samples will contain the correct lookup. If our attacker adjusts their attack with this process in mind, then they can for instance scan multiple regions of the Cache. If the current configuration is a 4-way configuration, then the attacker can scan for two potential locations per table lookup and combine the results.

In our hypothetical situation, where the attacker only has access to certain lanegroups and can not configure to the other Caches (while the AES library has the privilege to do so), we isolate a part of the lookups outside our attacked Cache. If however, the attacker has access to a setup where they do have access to all system resources, and the vulnerable round is split up over the individual Caches, then the attacker can only target the ”extra” Caches for a Prime+Probe Cache. These can theoretically have less noise if the a process execute with less memory accesses, or memory accesses that map less to the table address space, or even isolate the lookups by making sure no memory accesses are done in those contexts.

Trace Attacks
Trace attacks on implementations with four tables used on a round should be fully prevented, as the targeted lookups that potentially collide no longer share a Cache. Trace attacks that target rounds implemented with the same lookup table for every lookup will be hit like with the previously described final round collision attack, Where collision attacks on the ρ-VEX can ideally reduce the search space to 256 options, or 8 bits of uncertainty, splitting up these encryptions increases it to 32 bits of uncertainty, or $2^{32} = 4.3 * 10^9$ iterations of brute force search.

4.5.3. Implementation: Standalone Encryption

We first describe the implementation when a single context runs AES encryptions and the processor is potentially shared with other contexts or claims all 8 instruction lanes.

Before the AES encryption, we randomize the order of configurations to the 4 smallest Caches in the system (that would be 0x0888, 0x8088, 0x8808 and 0x8880 if no other contexts run).

If another context runs in the processor, the configuration words are chosen in a way that no illegal configurations are requested. There are certain options on possible configurations patterns. Lets illustrate this with an example.

We have our AES encryptions running in Context0, and another workload in Context1. We initialise this in configuration 0x1100, so context1 can theoretically keep running if Context0 configures to a smaller issue width (as 0x0811 and 0x8011 are illegal configurations). If we want to generate our pattern of splitting Context0 over all Caches, two of the configuration words are easily derived: those would become 0x1108 and 0x1180. Now, if we interrupt context1 to briefly allow context0 access to its resources, we can choose to let context1 run in the resources not selected by context0. This would however interfere with the Cache state for Context1 even further. We decide to stall Context1 during those cycles. From 0x1100 we will thus generate the configurations 0x0888, 0x8088, 0x1108 and 0x1180.

4.5.4. Implementation: 4 context AES Setup

Another approach is to run AES encryptions in a dedicated mode where all four hardware contexts run AES encryptions in parallel. This solves the issue of large Cache related performance costs associated with this Countermeasure, as all contexts will (partially) have the same data cached.

We implement ScatterRound Countermeasure on a setup of 4 parallel AES encryptions as followed:

1. Start all encrypting contexts, this can be done via synchronizing what block to encrypt and requesting the next one per context. We implement this as every context just generating random plaintexts in these experiments.

2. When the protected round is reached, the context requests the privilege to rotate, making sure no other round issues reconfigurations during this part of the algorithm.

3. When another active context is in this rotation mode, wait till it leaves this mode, and this context can claim privilege to rotate.

4. Randomly initialise the rotation direction to clockwise or counter clockwise.

5. Execute the AES round, rotate all active contexts after the first three columns.
6. After the final column, stay in the new configuration. Release the privilege to rotate.

We identify this implementation to have a couple beneficial characteristics:

- When encrypting a large amount of blocks, the execution will eventually synchronize in such a way that no stalls for waiting on entering the protected round should happen. As the rest of the encryption takes at least 9 times the execution time of the protected round.
- This implementation keeps the randomness aspect of the earlier discussed implementation ideas for this Countermeasure, through the combination of a random rotation direction, and not switching back to the default configuration but keeping the final rotation. This implementation thus also obscures in what Cache exactly which encryption happens, which can be beneficial against other types of attacks that do not rely on collisions.

We will estimate the cost of this Countermeasure. In [29], the design of the runtime reconfiguration is described. An example describes the best case delay, but explains that it is difficult to determine the configuration delay, as it depends on if the memory bus is currently accessed, the state of the Cache write buffers and the exact reconfiguration words.

It does describe the best case reconfiguration cost for the reconfiguration 0x0123 to 0x3210. This is 14 stall cycles for the main context that issued the reconfiguration, and 6 cycles for the other contexts involved. In the best case, we thus get stalls from 3 self issued reconfigurations and 9 reconfigurations issued by other contexts per encryption, this is 14*3 + 6*9 = 96 cycles overhead per encryption.

4.6. Conclusion

In this chapter, implementations of the Cache Attacks Prime+Probe, Evict+Time and Final Round Collision on the ρ-VEX are described. Then, three Countermeasure concepts proposed in Chapter 3 are worked out in three Countermeasure implementations called n-Lane, CacheSwap and ScatterRound. Practical details required to implement the Cache Attacks and Countermeasures are first introduced, describing the assumptions on reconfiguration privilliges, how to construct access attacks against the Cache architecture, how to determine reconfiguration words and how random reconfigurations as a Countermeasure are implemented in code. Our implementations of Evict+Time, Prime+Probe and Final Round Collision are described in detail. Then, three Countermeasure are introduced, describing their design, implementation, and potential effect on the attacks under certain circumstances. We first describe n-Lane: Noise via random configuration size variations. The amount of execution lanes is randomly changed during execution. The practical issue of losing cached data between configurations is discussed, and the potential for this Countermeasure to disturb the Cache state of the context is described. Finally the code implementation of the Countermeasure is described. We decide to test multiple configuration patterns possible within this Countermeasure, varying with what Caches are used and what reconfiguration percentages are used. Then, we describe *CacheSwap:* access noise via lanegroup swaps of two contexts. Another process is briefly swapped with the attacked process, so that the assumed Cache state is disturbed. The potential effects this Countermeasure can have on specific Cache Attacks is discussed, as it should influence all our attacks in unique ways. The importance of another context sharing the processor is highlighted. Finally we introduce the Countermeasure *ScatterRound:* preventing internal collisions via spreading operations over multiple Caches. This Countermeasure combines the potential of preventing collisions that are used for collision attacks and moving certain data lookups to a different Cache, by moving parts of the attacked rounds to isolated Caches on every encryption.

5

Results

In this chapter, the experimental results of the Cache attacks against the Countermeasures introduced in Chapter 4 are presented. Section 5.1 describes the experimental setup, synthesizing the ρ-VEX processor on the Digilent Genesys2 Development board. Section 5.2 analyses the raw performance of our AES implementation on the ρ-VEX and its speedup due to cached lookups. Section 5.3 gives an overview of the metrics that are provided in the results that show the performance of the Countermeasure against the attacks, and Section 5.4 gives and overview of the experiments that are done in the rest of the chapter. Section 5.5 gives the baseline results of the attacks. Section 5.6 does experiments on the influence of the shared memory access between multiple processes in a shared processor. Section 5.7 gives the experimental results of the n-Lane Countermeasure, Section 5.8 that of the CacheSwap Countermeasure and Section 5.9 of ScatterRound. Finally, a conclusion on the results is given in Section 5.10

5.1. Experimental setup

5.1.1. FPGA

FPGA's and HDL's

A Field Programmable Gate array (FPGA) is an integrated circuit that implements a reconfigurable logic circuit. A designer describes their logic design in a hardware description language (HDL). The two most commonly used languages are VHDL and Verilog. Synthesis tools like Xilinx Vivado are used to first Synthesis a circuit: translate the HDL code to a virtual circuit of logic gates. Then this synthesized design is implemented on a FPGA board, implementing the synthesized design with the resources available on the FPGA.

The main building blocks of the FPGA are configurable logic functions (LUTs) and register based storage grouped together on so called slices or logic blocks. Finally, configurable interconnect connects these slices together. Modern FPGA's also include certain hardwired function blocks, for functionality that is traditionally very inefficient if implemented in the standard slices. These include more complex computation blocks like multipliers, or Digital signal processing (DSP) slices and larger BRAM memories on Xilinx boards.

FPGA's find wide usage as a substitution of a dedicated ASIC chip. Common reasons to use FPGA's over ASICs are debugging functionality, saving cost for small volumes, research or academic purposes or in applications with runtime reconfigurable chip designs.

Genesys2

The FPGA development board used to instance the ρ-VEX processor is the Digilent Genesys2 [52], seen in Figure 5.1. The FPGA part on this board is a Xilinx Kintex-7 chip, XC7K325T-2FFG900C [53]. It has 326,080 logic cells. Consisting of 50,950 logic slices that can be configured to a maximum of 4,000 kB of distributed ram. It further contains 840 DSP slices and 16,020 kbit of BRAM.

The board includes 1 Giga Byte of volatile DDR3 memory and 16 Mega Byte of non-volatile flash memory. It has various connection ports for peripherals, including multiple USB connectors, HDMI,

AUX, FMC, VGA and microSD card connector. Only the DDR3 memory, USB UART connection and switches for debugging purposes are used on this board.

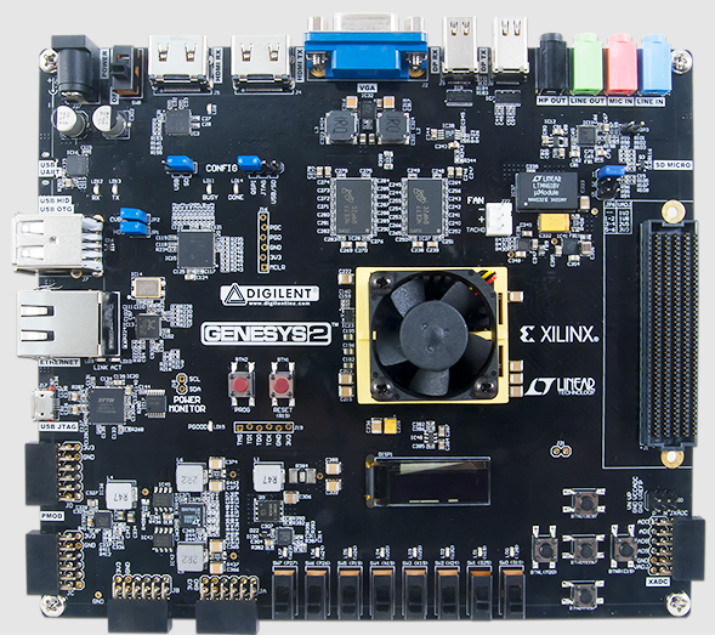


Figure 5.1: Genesys2 development board [52]

5.1.2. ρ-VEX setup

This section briefly describes how the ρ-VEX design is instanced on the FPGA.

Selecting system parameters

We will use the default parameters of the ρ-VEX presented in [29] and included in the v4.2 hardware design of the ρ-VEX. A brief summary of the selected parameters can be found in Table 5.1

ρ-VEX configuration	**config**	**details**
Lanes	8	Instruction lanes
Lanegroups	4	Pairs of two lanes
Contexts	4	Hardware contexts with register files
Bundle alignment	2	Instruction bundle minimal size, using stop bits
LIMMH from next	yes	Long immediate syllable forwarding
LIMMH from previous	no	Long immediate syllable forwarding
Lanes with multipliers	all	lanes that can execute multiplication instructions
Lanes with memory units	0,2,4,6	Lanes that can acces system memory
Instruction Cache	4x16kB	Up to 64kB in largest configuration
Data Cache	4x8kB	Up to 32kB in largest configuration

Table 5.1: ρ-VEX configuration

These settings are mainly the default parameters provided with the design. We adjusted the Cache parameters to fit the requirements for the experimental Cache attacks:

- **Data Cache:** at least enough space for all AES table data is required to prevent Cache data being evicted by the AES process itself. With the AES implementation used, 4 tables of 1024 bytes and 1 table of 256 bytes are used for a total of 4352 bytes. As we can only pick a power of 2 Cache lines of 4 bytes, the Cache size is set to 2^{11} lines, 8kB per lanegroup thus 32 kB total. We guarantee no Cache size penalty and thus no evictions of the AES data by its own lookups in the smallest configuration that uses only one Cache.

- **Instruction Cache:** with the instruction Cache size, we want to prevent that the smaller configurations have a large performance penalty in comparison to the larger configurations, so at least the full AES code should fit into the instruction Cache. it was found out experimentally that an instruction Cache size of 2^9 lines (16kB) per lanegroup was sufficient.

Full system

The ρ-VEX core is integrated into two types of full computing systems.
1. The first is a standalone setup. Where the system memory is emulated into the FPGA DRAM slices. The access time delay for lookups to this emulated memory can be configured to emulate the delay a external memory would cause. This system also implements a connection to the debug bus over USB UART.
2. A GRLIB based system. Figure 5.2 show a schematic of this system. This system is based on the LEON3 example design for the ML605 development board [54] found in the GRLIB library [55]. It substitutes the LEON3 processor with the ρ-VEX. Figure 5.2 shows a general schematic of this design. This system allows peripherals compatible with the GRLIB standard library to be connected to the core. Most importantly, this allows external (DDR3) memories to be connected to the core.

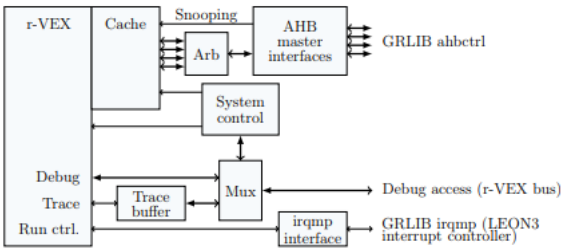


Figure 5.2: Block diagram of the GRLIB processing system. [29]

Since we want to have realistic memory behaviour, and sufficient emulated memory size can not be reasonably achieved with the DRAM on the Genesys2, which is only 16Mbit, the GRLIB based system was selected. This system was ported by using the design presented in [29]. Figure 5.3 shows a schematic overview of the system. Major adjustments made were the removal of unused connections (Ethernet, PCIe), adjusting clocking to match the board and creating the memory interface to work with the DDR3 memory on the Genesys2 board. In the synthesis tool Xilinx Vivado, the build in MiG (Memory interface Generator) was used to generate the MiG component that connects the DDR3 pins to the FPGA and creates an interface with it. A ahb2mig component was present in the example design, which converts the MiG interface to the AHB bus. This design was adjusted to match the changed interface for the Series 7 equivalent. The design was synthesized and implemented in Xilinx Vivado. The system runs on a clock speed of 20MHz.

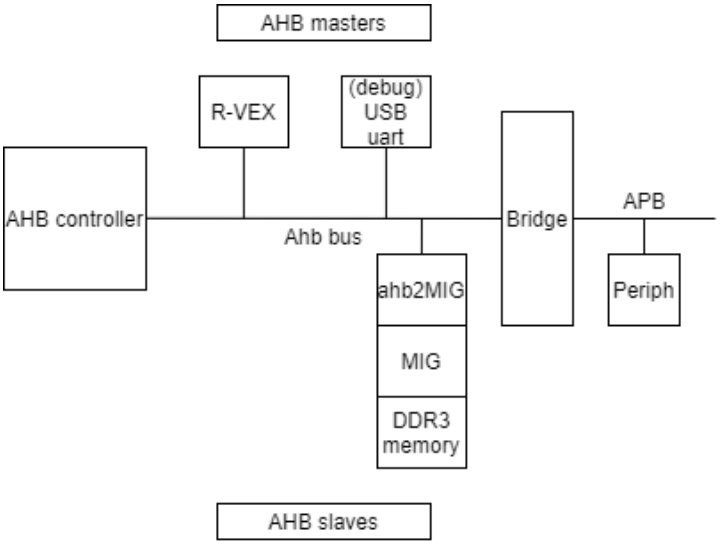


Figure 5.3: Schematic of the GRLIB based processing system on the Genesys2

5.1.3. Interfacing setup

Because no support to directly dump memory data to a file was found within the debugger toolkit, a small pipeline was set up to get memory dump data from console output to a *xlsx* format. The pipeline for the memory dump goes via the following steps:

1. The console output in unix is stored in a *txt* file
2. The textfile is loaded into python
3. Python removes characters used for the command line layout, keeps raw data of addresses and data in a data structure.
4. Python substitutes columns that indicate that certain memory ranges are the same with copies of that data. These are originally not printed by the debug tool.
5. Python splits the lines into words, converts them from hexadecimal to integer and stores them in a *xlsx* file.

This will result in a 4xn table of data in a *xlsx* file. The offline attack part in MatLab will interpret this data, and further group the data based on the known layout of the measurement data for that specific attack.

5.1.4. Software setup

We create two setups by compiling two sets of code. One to run an isolated attack process on the ρ-VEX, and a second setup where a shared processor is emulated with one other context executing a random workload.

Standalone setup

The OpenSSL T-table implementation of AES-128 was ported to the ρ-VEX for this thesis. Instead of implementing the final round with transformations on the main 4 T-tables, like in the current OpenSSL implementation, the final round is implemented using the AES S-Box implemented with a fifth table, T4. This table has 256 32-bit entries, with the S-box values at all 4 byte offsets. This implementation allows for the final round collision attack to be tested efficiently. This has a small benefit to attacks that use earlier rounds, as 4 random lookups per table are no longer done during an encryption when compared to the OpenSSL implementation. The attackers code is compiled together with this AES-128 implementation. The memory location of the Table data is initialised at a memory offset that is known to the attacked. The AES key expansion is first initialised, after which the attacker code starts, which invokes the AES encryption function at its own initiative. The ρ-VEX is initialised in the configuration 0x0088 by default, and thus runs in 4-way mode.

Shared processor setup

In order to simulate a workload sharing the processor with the encrypting context, we run a set of PowerStone benchmarks [56] in randomized order on a context in parallel to the encrypting context. The list of PowerStone benchmarks, bcnt, blit, compress, crc, des, engine, fir, g3fax, jpeg, pocsag, qurt, uxbqsort and v42, have been previously ported to the ρ-VEX. We initialise the first context, context0 to run the same attack process as in the standalone setup, and initialise context1 to run a random selection of the PowerStone Benchmarks until context0 finishes. Between each benchmark, we add a set amount of stall cycles, so that the amount of workload on that context can be varied. The processor then runs in the configuration 0x1100 by default.

5.2. AES performance on the ρ-VEX

Although performance penalties of bypassing the Cache completely in literature have been stated to be up to a factor 100, we would like to get an indicator of the performance of the AES implementation used in these experiments.

The Cache miss penalty for a single lookup, without noise from the shared memory bus, was measured to be 9 cycles.

The instruction Cache has 8 words per line. Reloading such a line take an additional cycle per word, thus 16 stall cycles. If variable length bundles are enabled, then two Cache lines can result in misses for a single instruction bundle, resulting in 32 stall cycles if the bundle is not aligned with the instruction Cache lines.

If other contexts are executing, then the delay on a read or write from the memory can be increased, as the ρ-VEX only allows one context at a time to access the memory. If a context is already being serviced, the others have to wait till they gain access to the memory. If a write is being handled, Cache lookups have to wait for it because it is required for Cache consistency [29].

Cache misses in mainstream CPU's have relatively more penalty on a single Cache miss. This is partly caused by longer Cache line sizes. A Cache line size implementation on the ρ-VEX, of 16 32-bit words per line, would increase the stall cycles on a Cache miss in the ρ-VEX to 21 cycles if these all are sequentially transferred.

These fast memory lookups are relevant to keep in mind with Countermeasures that (purposely) causes Cache misses, as the measured performance decrease can be misleading if not held against the worst case execution (data Cache bypasses) for the current platform. Thus, Countermeasures that cause Cache misses will have to be put into perspective by using these worst case execution times, when fully bypassing the Cache for vulnerable lookups.

AES implementation performance

During testing, the performance of smallest configuration was disproportionate to what was expected based on assigned lanes, this was caused by a high amount of instruction Cache misses. Because in this research we want execution in the tiniest configuration to be efficient, we decided to rewrite the code to be more compact, to lower the chance of disproportional performance costs because of the instruction Cache. The original implemented the rounds as manually written out lines of all rounds. In our implementation, 8 rounds were written more compactly in a for-loop of two rounds that repeats 4 times with parameterised indexes for the roundkeys. This means the effective amount of execution cycle is a bit higher than in the OpenSSL version, but it requires less instruction Cache and we measured better performance that comes closer to the difference someone would expect when switching between configuration size.

Table 5.2 shows the measurement results we did measurements on the performance in all 3 possible issue widths of the ρ-VEX. The ρ-VEX has the ability to disable the data Cache and bypass it directly to the memory. The instruction Cache will remain active in this mode. These results can also be seen in this table.

AES performance	Baseline	Disabled data Cache	Disabled data Cache % extra cycles
2-way	912	3966	+435%
4-way	587	3433	+585%
8-way	323	2334	+650%

Table 5.2: Baseline performance of our AES implementation for all issue widths. Compared to function call with data Cache bypass.

Cost of disabling Cache in targeted rounds
In order to get a grasp of how much performance we waste in our harshest Countermeasure possible; bypassing the Cache for all lookups that are targeted by our attacker, we compute how much overhead costs we expect if we bypass these lookups:
Although in the setup we were not able to disable the data Cache directly in code (only via the debug bus), we can compute theoretical performance costs if we disable the Cache in a AES round:
Perfect implementation: Only the 16 table lookups for a round are disabled. A non-cached lookup adds 9 cycles so the cost is 144 cycles per round.
Full round implementation: If the designer of the code has no access to directly disable per individual lookup, but has to do the disabling for a full round then 20 lookups are bypassing the Cache, resulting in 180 extra cycles.

The costs for the most drastic approach, disabling the Cache for targeted lookups, can be found in Table 5.3. We make the distinction between the possibility of bypassing the Cache for the Table lookups only or for all lookups in a round. These computations are for a 4-way configuration, as that is the one we will use as the baseline configuration in our experiments.

Cache disabling costs	Baseline	1 round, Tables	1 r, Round	2 rounds, Tables	2 r, Round
Cycles 1 AES encryption, 8way	323	467	503	611	683
% execution time, 4way	-	+44.6%	+55.7%	+89.2%	+114.6%
Cycles 1 AES encryption, 4way	587	731	767	875	947
% execution time, 4way	-	+24.5%	+30.7%	+49%	+61.3%
Cycles 1 AES encryption, 2way	912	1056	1092	1200	1272
% execution time, 4way	-	+15.8%	+19.7%	+31.6%	+39.5%

Table 5.3: Costs for disabling the Cache in targeted rounds, on the ρ-VEX in a 4-way configuration

5.3. Metrics

Key rank or solution rank as a function of traces
For Prime+Probe and Evict+Time these subsolutions directly reveal the key byte values, in these attacks we will thus refer to this metric as the average keyrank. In the Final Round Collision these measurement are the values of xor's between key bytes, and thus we express this metric as the average subsolution rank. We choose to look at this metric for this attack as it most accurately shows the influence of Countermeasures on the measured data.

Correct subsolutions as a function of traces
In Prime+Probe and Evict+Time, a subsolution is the most likely candidate access, and thus table lookup, for a chosen key byte index and a chosen plaintext byte on that same index. This is one sub measurement to target a specific key byte. We consider this to carry more information than presenting how many key bytes were retrieved correctly, as this depends on the method in which the attacker decides to do the post processing. This method will be the most neutral way to present the influence of the Countermeasures on the measured data used for the attack.

Relative processor performance
The relative performance of the AES encryption code will be measured as part of the Evict+Time attack. This is expressed in a percentage increase or decrease of the average encryption time, when executing in the same baseline configuration. The measurements from Evict+Time are used, because this execution is the most accurate to regular execution conditions of the three attacks, as Prime+Probe and Final Round Collision clear all cached T-Table data during their attacks before encryption.

Why it is also ideal to combine these measurements in the Evict+Time attack is to make sure the overhead costs closely match with the measured influence on attack effectiveness, in order to compare both. If these measurements were done in isolation, then for instance the instruction Cache usage of the process that does the measurement individually can cause different levels of interference from the Countermeasures and thus also different performance costs.

In the setup with a shared processor, the performance of the PowerStone benchmarks [56] are measured. These are also reported as relative performance when compared to baseline measurements in the same default configuration.

5.4. Overview of Experiments

For every Countermeasure there are certain variations the can be done during the experiment. These include:

- Changing the fundamental reconfiguration pattern used; adjusting what contexts are used and in what ratio configurations occur and if they are randomly selected or have set order.
- Changing probabilities for Countermeasure that have a chance to be triggered.
- Cache state variations at the start of encryption.
- Shared processor setup vs isolated process.

We will test the Countermeasures: *n-Lane*, where we randomly change the amount of execution lanes assigned to our encrypting context, *CacheSwap*, where we randomly swap executing contexts briefly from resources and thus Cache used, and *ScatterRound*, where vulnerable rounds are split among multiple Caches. These are tested against the studied attacks Prime+Probe, Evict+Time and Final Round Collision.

At the end of the experiment sets on the Countermeasure, sets C, D and E, there is a small section where we summarize our main results of the experiment: the effect on the attack effectiveness, the overhead costs and additional findings with regards to the different implementation and execution scenarios possible with the Countermeasure. We will then discuss the findings of the experiment set. Since the experiment sets can be quite lengthy, we recommend the reader to first read this summary and discussion section, to then decide on what experiments to look at in detail.

Experiment Set A: Cache Attacks against AES in a noiseless processor
First, the performance of the three studied Cache Attacks in a noiseless processor are measured. The attack, that invokes the encryptions itself, is the only running process in the processor. All other resources not used are disabled. The experiments are:

- **A.ET.1)** unprotected Evict+Time
- **A.PP.1)** unprotected Prime+Probe
- **A.FR.1)** unprotected Final Round Collision

Experiment Set B: Cache Attacks against AES in a shared processor
The second experiment set executes the attacker process, that is isolated in Experiment Set A, on half of the processor resources, in 4-way mode. In the other half of the processor, a second context is executing in 4-way mode. This context executes randomly selected PowerStone benchmarks [56], with a set amount of stalled cycles in between those benchmarks to emulate different workload conditions.

- **B.ET.1)** unprotected Evict+Time, shared processor
- **B.PP.1)** unprotected Prime+Probe, shared processor
- **B.FR.1)** unprotected Final Round Collision, shared processor

Experiment Set C: Cache Attacks against n-Lane protected AES
Experiment set C shows the results of the n-Lane Countermeasure, that generates noise via reconfiguration size variations.

- **C.ET.1)** n-Lane against Evict+Time, reconfiguration patterns
- **C.ET.2)** n-Lane against Evict+Time, reconfiguration percentages
- **C.ET.3)** n-Lane against Evict+Time, reconfiguration interval
- **C.ET.4)** n-Lane against Evict+Time, trace size requirement
- **C.ET.5)** n-Lane against Evict+Time, shared processor
- **C.PP.1)** n-Lane against Prime+Probe
- **C.FR.1)** n-Lane against Final Round Collision, reconfiguration percentages
- **C.FR.2)** n-Lane against Final Round Collision, shared processor

Experiment Set D: Cache Attacks against CacheSwap protected AES
Experiment set D shows the results of the CacheSwap Countermeasure, that generates access noise via lanegroup swaps of two contexts.

- **D.ET.1)** CacheSwap against Evict+Time, Cache state
- **D.ET.2)** CacheSwap against Evict+Time, targeted round effect
- **D.ET.3)** CacheSwap against Evict+Time, reconfiguration percentages
- **D.ET.4)** CacheSwap against Evict+Time, shared processor
- **D.PP.1)** CacheSwap against Prime+Probe, reconfiguration percentages
- **D.PP.2)** CacheSwap against Prime+Probe, shared processor
- **D.FR.1)** CacheSwap against Final Round Collision, reconfiguration percentages
- **D.FR.2)** CacheSwap against Final Round Collision, shared processor

Experiment Set E: Cache Attacks against ScatterRound protected AES
Experiment set E shows the results of the CacheSwap Countermeasure, that obfuscates Cache accesses via spreading operations over multiple Caches

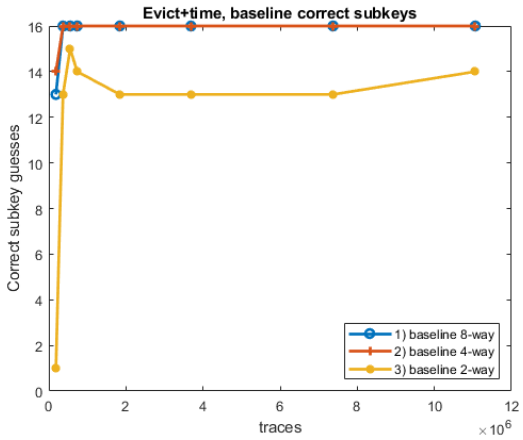
- **E.ET.1)** ScatterRound against Evict+Time
- **E.PP.1)** ScatterRound against Prime+Probe
- **E.FR.1)** ScatterRound against Final Round Collision, standalone execution
- **E.FR.2)** ScatterRound against Final Round Collision, shared processor
- **E.EX.1)** ScatterRound performance in a dedicated execution mode

5.5. Experiment Set A: Cache Attacks against AES in a noiseless processor

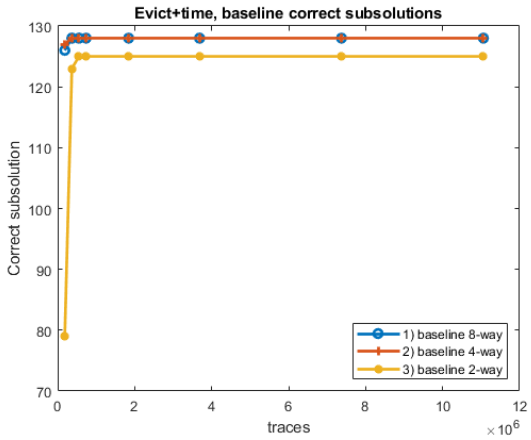
We test our attacks in a noiseless environment. We only have one context active on the processor. All resources not used by our active context are idle, meaning there is no interference on memory accesses from the shared memory ports. Other interference is kept to a minimum: the encryptions and attackers code come directly from the same compiled files; there is no operating system in between and the tasks are executed without other tasks getting scheduled.

5.5.1. A.ET.1) unprotected Evict+Time

The evict time attack is done on all 3 configuration sizes. Figure 5.4(a) shows that our attack is successful in retrieving the full key after 10 iterations per plaintext-targetkey-eviction target combination, which means we achieve these results after 368640 encryptions timed.



((a)) Evict+Time Baseline subkeys correct



((b)) Evict+Time Baseline subsolutions correct

Figure 5.4: Evict+Time Baseline results

We see that for a 2-way configuration, the attack fails to retrieve the all correct subkeys. To analyse further, we also plot the amount of plaintext-targetkey combinations correct (128 total), which are so called ”subsolutions” used to retrieve the correct key. Note that this does not necessarily mean the correct keybit guess, which could still be correct depending on how we guess the key (pattern vs direct calculation, as discussed in Section 4.2.1) but indicates if the maximum value is on the eviction address we expect it to be. In theory, 32 correct subsolutions (2 per subkey) should be enough to confidently guess the correct key in these attacks, if the other samples yield random maximums on the other values because of measurement noise. Yet we will visualize the performance of the attack via the amount of correct subsolutions to paint a more accurate image of how much useful information we retrieve from our samples, as opposed to only stating how many subkeys are correct in our attack. Figure 5.4(b) shows the correctness of the subsolutions as a function of the amount of traces used.

We observe that the incorrect subkeys are caused by only 3 incorrect subsolutions. These should be caused by Cache interference from its own calling procedure that is only caused in the 2-way configuration, because more addresses are mapped onto the same lines in the smaller configurations. Lookups to either the stack or locations of the plaintext, expanded key or software stack can cause evictions to the table data.

In Figure 5.5 the average subsolution ranks for the trace counts are plotted. For every subsolution, we determine how the correct samples rank among the potential candidates. In case of Evict+Time this means that we rank the samples from highest to lowest average time, and check on what position the correct sub-solution is ranked. We see the average keyranks approach 1 for the measurements on the 2-way configuration, meaning the wrong samples are only off by a small amount, verifying that indeed natural evictions make it impossible to directly measure these samples.

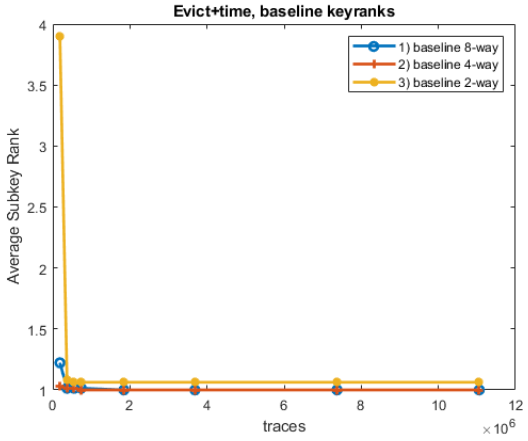
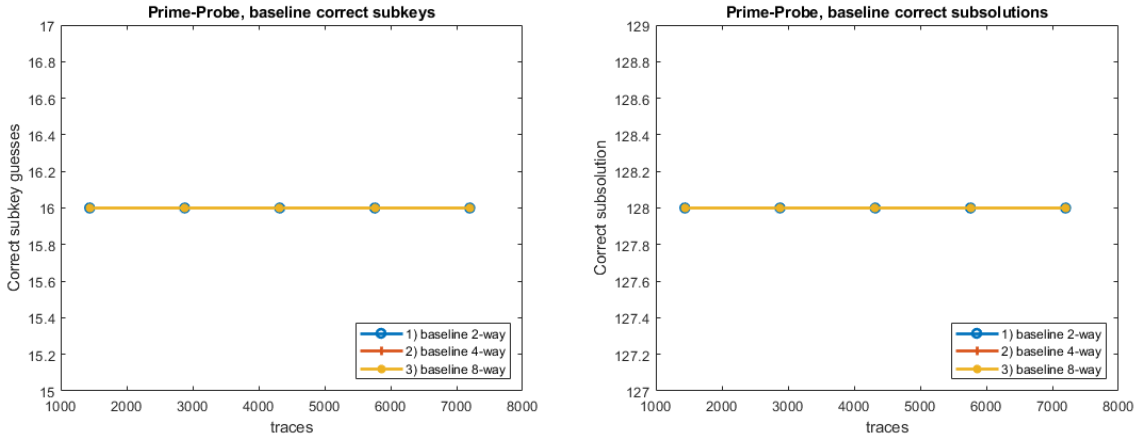


Figure 5.5: Evict+Time Baseline average keyranks

5.5.2. A.PP.1) unprotected Prime+Probe

The Prime+Probe attack yields the same measurement results as with the Evict+Time attack, but based on different measurement samples. We apply the same result analysis as we did with Evict+Time. In Figure 5.6(a) the amount of subkeys correctly found are plotted for increasing trace counts. We test this for all three configuration sizes. We found that at an arbitrary low value of 5 encryptions per plaintext+target subkey combination, enough samples were acquired to successfully execute the attack in all configurations. Figure 5.6(b) shows the plot of the subsolutions, which also are all correct for this low sample count. We conclude on 720 encryptions as the baseline sample requirement.



((a)) Prime+Probe Baseline correct subkeys

((b)) Prime+Probe Baseline correct subsolutions

Figure 5.6: Prime+Probe Baseline results

Figure 5.7 shows the plot the average keyrank of the subsolutions. We see that these also perfectly match with what we expect: all subsolutions are correct, and thus our average keyrank is 1.

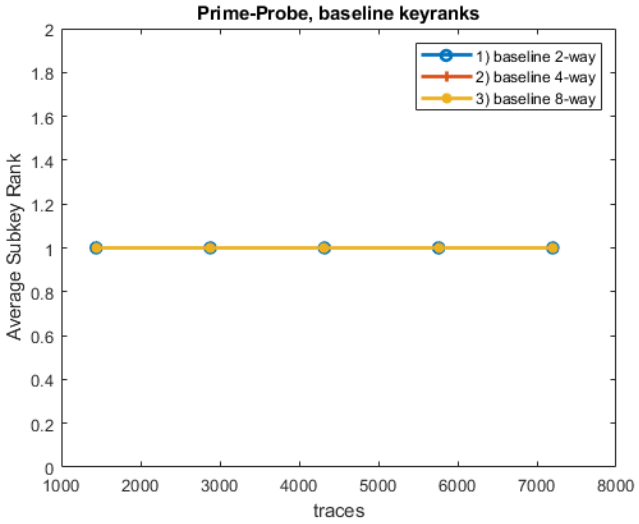
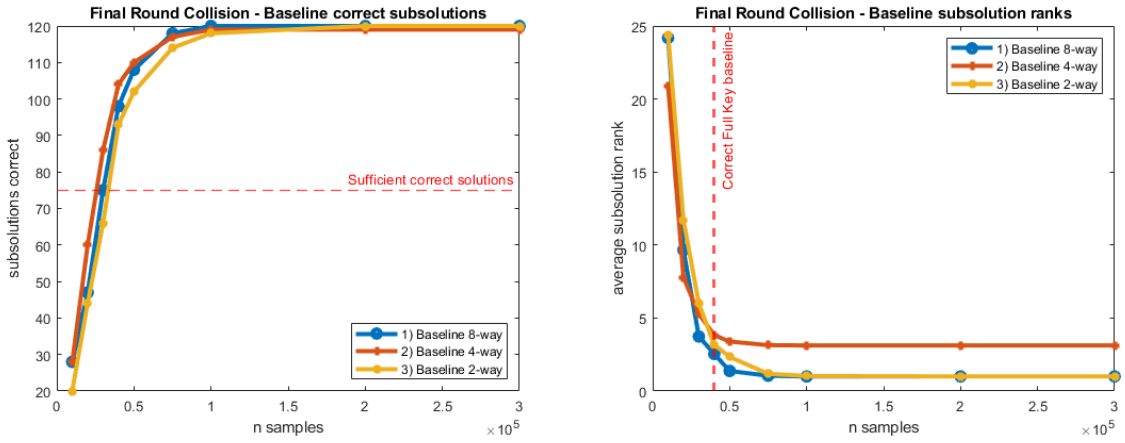


Figure 5.7: Prime+Probe Baseline keyranks

5.5.3. A.FR.1) unprotected Final Round Collision

We perform the Final Round Collision attack on our random plaintexts. Figure 5.8(a) shows the amount of subsolutions that are correct. Subsolutions are the found values of the 120 XOR inequalities between ciphertext outputs based on the value that has the minimum execution time. Figure 5.8(b) shows the average rank of the 120 inequalities for the amount of traces used in the attack.



((a)) Final Round Collision Baseline correct subsolution

((b)) Final Round Collision Baseline subsolution ranks

Figure 5.8: Final Round Collision Baseline results

Since the measurement results do not directly translate to a single key guess but 256 key candidates, and because these are not directly translated as individual subkey guesses but combinations, we will not include the correct amount of subkeys as a result of this attack.

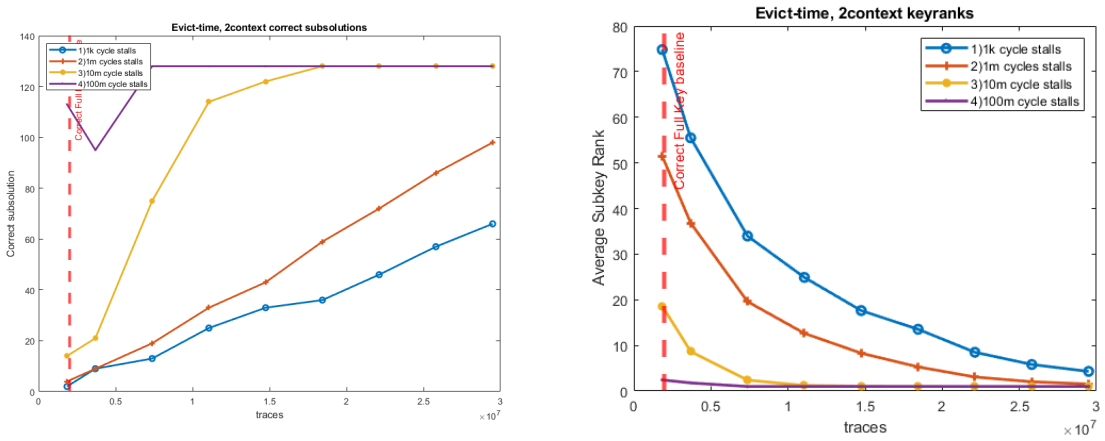
Instead, we only take a look at how many subsolutions are correct when rating the success of the attack in further experiments. We found experimentally that the point on which the correct candidate is retrievable is around 75 correct subsolutions on average. A small majority of correct subsolutions means that the correct guesses for the bit candidates will be the most likely candidates, satisfying the majority on the inequalities. All attacks have enough samples to retrieve the full key around 40000 samples. Just over 100000 samples are required to have correct measurement data on all subsolutions. The experiment on the 4-way configuration showed that 1 subsolution was not correct in this experiment.

5.6. Experiment Set B: Cache Attacks against AES in a shared ρ-VEX processor

In this section, the results from the attacks performed on a ρ-VEX system executing a second workload in half of the processor resources are presented. The attack process and AES runs in context0 on a 4-way configuration, and a context executes random PowerStone benchmarks runs on context1 in a 4-way configuration. Because the multiple hardware contexts share memory access resources, and have to wait on each other if the resources are occupied, influence on the performance of the attacks is expected. Because some Countermeasures rely on the fact another process is executing, these measurements are also done to get a reference on how much the attacks are influenced by the resource sharing itself.

5.6.1. B.ET.1) unprotected Evict+Time, shared processor

Figure 5.9 shows the results of the Evict+Time attack on the shared processor. The workload on the second processor is varied with stall cycles in between benchmarks, varying between 10^3 and 10^8 cycles. Table 5.4 shows the measured average performance cost for the throughput of the AES on our main context. The random extra cycles during encryption prove to be very disruptive to the Evict+Time attack, which relies on timing measurements of a full encryption which includes hundreds memory accesses, leaving room for many moments the other contexts can disturb the process via their own accesses.



((a)) Evict+Time shared processor correct subsolutions

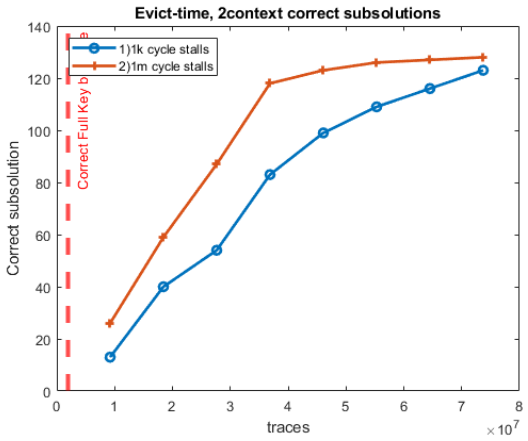
((b)) Evict+Time shared processor subkey ranks

Figure 5.9: Evict+Time shared processor results

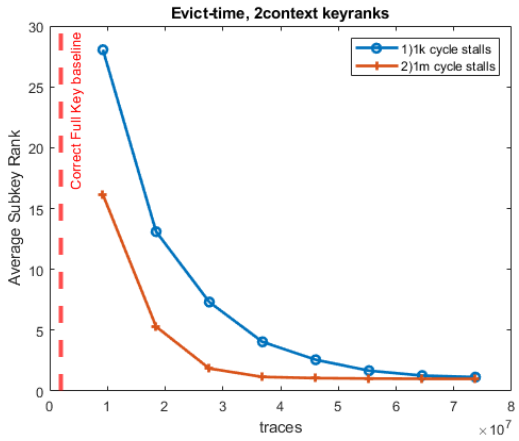
Evict+Time 2Context	1) 1k	2) 1m	3) 10m	4) 100m
% average execution time AES	+3.4%	+0.8%	+0.1%	+0.01%

Table 5.4: Additonal execution time for the attacker + victim context when a second context shared the system

Figure 5.10 shows the results for a higher amount of traces for the parallel workloads with 1000 and 1 million stalls, since the attacks were found to not be successful yet for the amount of traces in the experiments in Figure 5.9. We see that the influence is rather insignificant for low workloads. We see the requirement of traces for all correct subsolutions go from a factor of 2 for the lowest workloads, to a factor up to 200 for the highest workload.



((a)) Evict+Time shared processor correct subsolutions

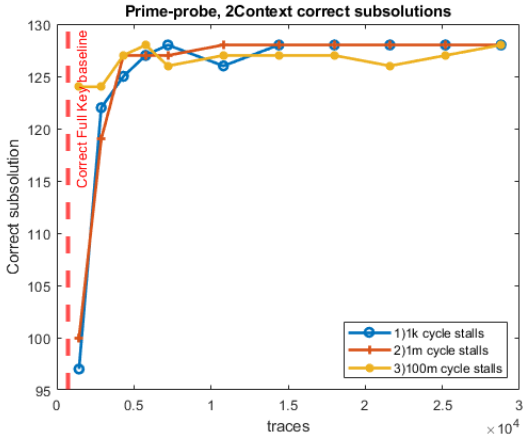


((b)) Evict+Time shared processor subkey ranks

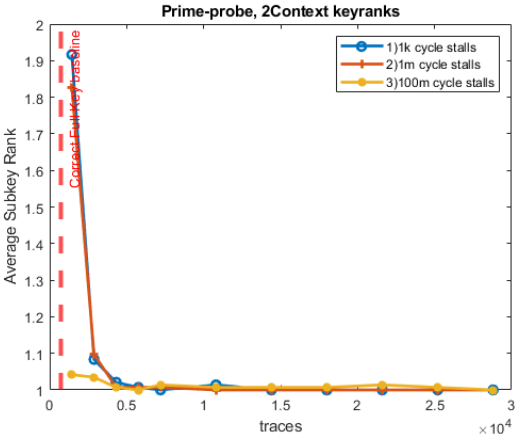
Figure 5.10: Evict+Time shared processor, high workload increased sample count

5.6.2. B.PP.1) unprotected Prime+Probe, shared processor

Figure 5.11 shows the results of the Prime+Probe attack on the shared processor. We see a much smaller influence on the effectiveness of the attack. It takes up to 20 times more samples to fully stabilize on having all correct subsolutions for the highest workload. However, we see that the samples are only distorted by a small amount around the trace count of our baseline. They average keyrank starts at only 2, and already 96 subsolutions are measured correctly (75%). This good performance is partly contributed to the way we implemented the attack, which does not take binary conclusions on hits/misses based on access times of a memory lookup, but averages the total time it took to access a Cache line. Based on the processing technique, this could already result in the correct full key retrieved based on this sample count. We thus conclude that the influence of the shared processor on the effectiveness of the Prime+Probe attack is negligible.



((a)) Prime+Probe shared processor correct subsolutions

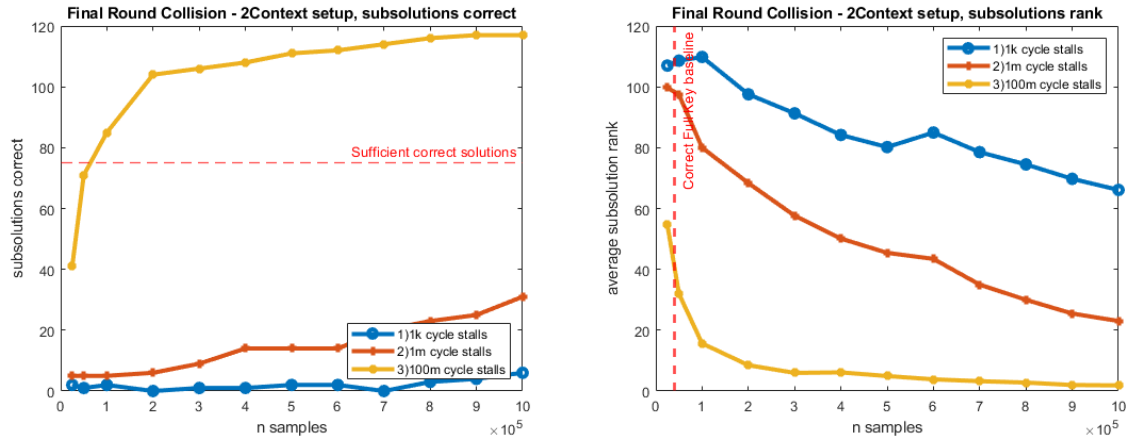


((b)) Prime+Probe shared processor subkey ranks

Figure 5.11: Prime+Probe shared processor high workload increased sample count

5.6.3. B.FR.1) unprotected Final Round Collision, shared processor

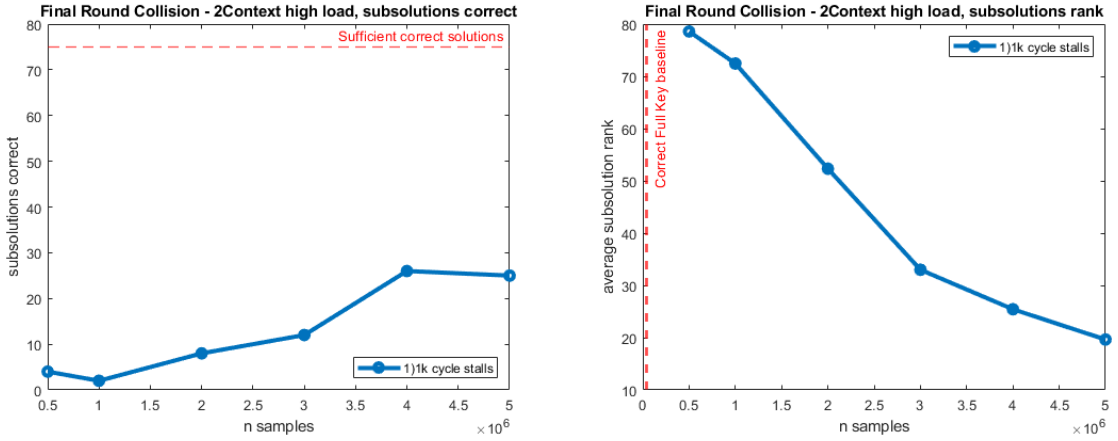
Figure 5.12 shows the results of the Final Round Collision attack on the shared processor. We roughly need 3 times as many traces to do a successful attack for the lowest workload. and the other two workloads tested are not solvable yet for 10^6 samples, already a factor 25 over the baseline.



((a)) Final Round Collision shared processor correct subsolutions ((b)) Final Round Collision shared processor subkey ranks

Figure 5.12: Final Round Collision shared processor results

We repeat the attack against the system shared with the highest workload in Figure 5.13, for five times as many samples. We do see the subsolution rank further decrease. We estimate the amount of traces required to be at least $8 * 10^6$ and thus at least 200 times as many traces as the baseline attack are required, similar to the Evict+Time attack in a shared processor for this same workload.



((a)) Final Round Collision shared processor correct subsolutions ((b)) Final Round Collision shared processor subkey ranks

Figure 5.13: Final Round Collision shared processor, high workload increased sample count

5.7. Experiment Set C: Cache Attacks against n-Lane protected AES

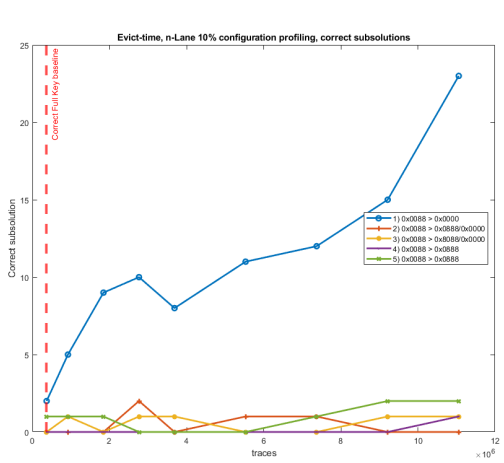
5.7.1. C.ET.1) n-Lane against Evict+Time, reconfiguration patterns

The first experiment done on the n-Lane Countermeasure is to see the difference between the variations done on the reconfiguration pattern that is used in the n-Lane Countermeasure. This is tested against the Evict+Time Attack. The probability of a reconfiguration is set to 10%. From the main configuration, 0x0088 we can either configure to larger configuration 0x0000, or smaller configurations 0x8088 and 0x0888. We will test for multiple variations of switching between these configurations.We also tweak the ratio between the amount of reconfigurations to the larger and smaller configuration, aside from the default 50/50 option. We test the following patterns:

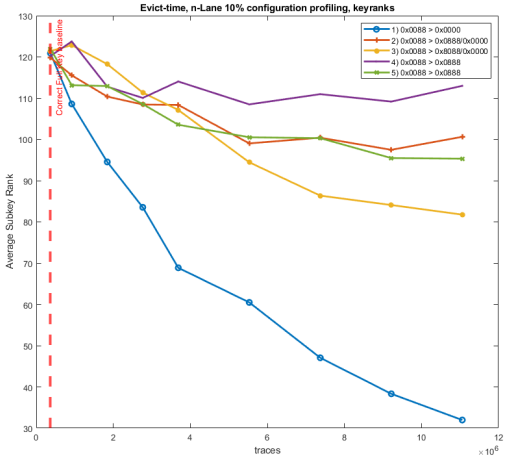
1. Only to larger configuration 0x0000

2. Both larger and smaller configuration, 0x0888 for the smaller configuration
3. Both larger and smaller configuration, 0x8088 for the smaller configuration
4. only smaller configuration 0x0888
5. only smaller configuration 0x8088
6. experiment 2. with 66.6% chance to configure larger and 33.3% to configure smaller
7. experiment 3. with 66.6% chance to configure larger and 33.3% to configure smaller
8. 3.33% chance to configure smaller to 0x0888 only

The results of experiment 1. to 5. are seen in Figure 5.14. The results of experiment 6. to 8. in Figure 5.15. Table 5.5 shows the measured relative slowdown of AES encryptions in these experiments.

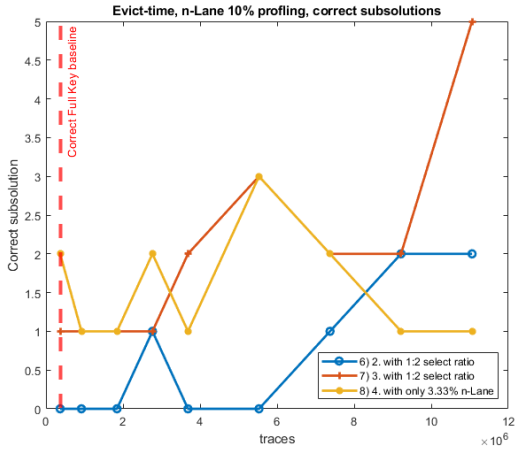


((a)) Evict+time n-Lane 10% patterns subsolutions

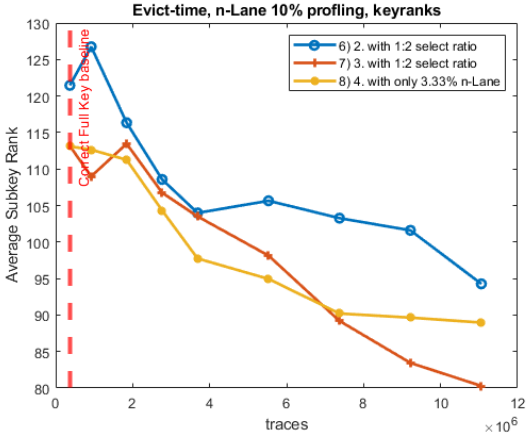


((b)) Evict+time n-Lane 10% patterns subkey ranks

Figure 5.14: Evict+Time n-Lane 10% patterns 1



((a)) Evict+time n-Lane 10% patterns subsolutions



((b)) Evict+time n-Lane 10% patterns subkey ranks

Figure 5.15: Evict+Time n-Lane 10% patterns 2

n-Lane configurations	1)	2)	3)	4)	5)	6)	7)	8)
% average execution time AES	-5.1%	+8.2%	+4.1%	+19.1%	+12.3%	+4.6%	+1.2%	+6.73%

Table 5.5: overhead costs of the reconfiguration patterns in Figure 5.14 and Figure 5.15

We summarize our findings in these experiments:

1. **The random reconfigurations to smaller configurations add more noise:** this can be seen through comparing experiments 1, 4 and 5 with each other. The chance to configure is constant between these experiments, but when configured to the smaller configurations we see that the loss of cached data and potential extra evictions add extra noise.

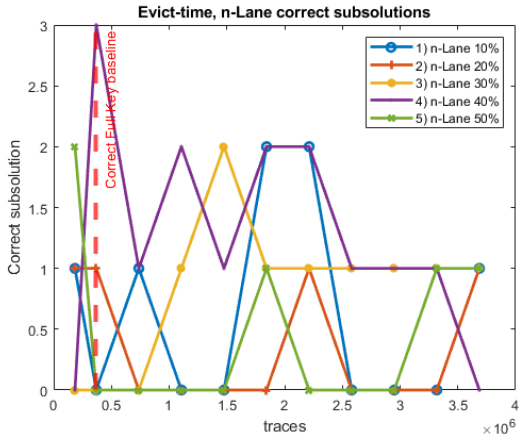
2. **There is a smaller configuration that suboptimal for performance, but adds more noise:** When comparing experiment 2 with 3, 4 with 5 and 6 with 7, it is observed that the configuring to configuration 0x0888 adds more noise than 0x8088. In Table 5.5 it is also seen that using this configuration reduces AES performance more than the other.

3. **Both smaller and larger configuration together contribute to noise:** Experiment 8 tests 3.33% chance to reconfigure to only smaller configurations. Adding the chance to also configure larger in experiment 6 adds extra noise.

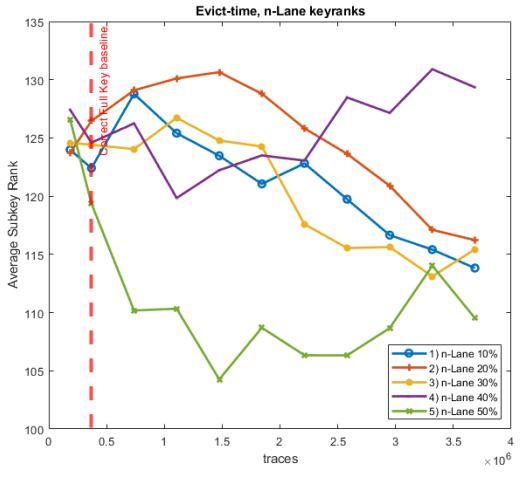
4. **A balanced ratio in the reconfiguration pattern is possible that still adds measurement noise:** We show with the reconfiguration ratio used in experiments 6 and 7 that we can adjust the ratios between the configurations, and eventually achieve a point where we do not decrease the throughput. We do however decrease the performance of other contexts by stalling them for our larger configuration. We can even improve throughput while adding noise, as seen in experiment 1.

5.7.2. C.ET.2) n-Lane against Evict+Time, reconfiguration percentages

From the previous configuration testing in Experiment C.ET.1), Section 5.7.1, the default configuration pattern of selecting both larger and smaller configurations with equal probability is chosen. The smaller configuration that has the most interference, the pattern of experiment 2, is selected as the configuration pattern to further experiment on. In these experiments, we look at the behaviour of higher reconfiguration probabilities for this pattern. In Figure 5.16 probabilities from 10% to 50% are tested. In Table 5.13, the relative AES performance is given. Up to around a 30% reconfiguration chance, the performance is above that of bypassing the data Cache for the lookups in a single round.



((a)) Evict+time n-Lane percentages subsolutions



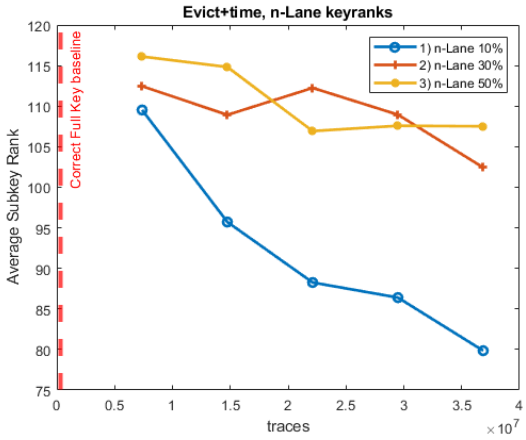
((b)) Evict+time n-Lane percentages subkey ranks

Figure 5.16: Evict+time n-Lane percentages

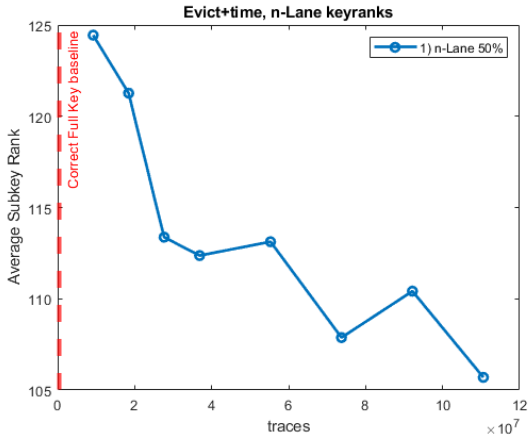
Evict+Time n-Lane percentages	1) 10%	2) 20%	3) 30%	4) 40%	5) 50%
% average execution time AES	+8.2%	+16.0%	+26.1%	+33.9%	+41.0%

Table 5.6: Overhead costs of the reconfiguration percentages in Figure 5.16

Because the sample size in Figure 5.16(b) is not sufficient to see if the general keyranks are decreasing for larger sample sizes. We increase the sample size by a factor 10 and do the attack on 10%, 30% and 50% n-Lane in Figure 5.17(a). Because for n-Lane 50% it still is unclear if eventually the key rank starts decreasing with more samples, we increase the sample size once again and repeat for n-Lane 50% in Figure 5.17(b). Between the percentages, the decline rate decreases for higher percentages. The amount of traces required for a successful Evict+Time attack against the AES protected by 50% n-Lane seems to be enormous. The cost on the throughput of AES is 41%, but is still below the cost avoiding the use of Cache in two AES round, while it can be effective against attacks that target either round. A bottleneck however, is the performance cost to the rest of the system, as this would stall the other contexts in the processor during 25% of the encryptions.



((a)) Evict+time n-Lane percentages subkey ranks



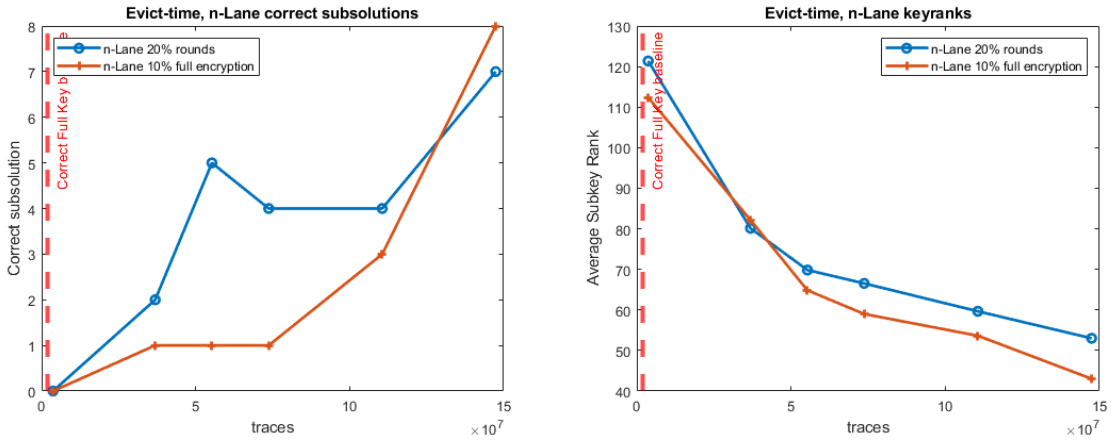
((b)) Evict+time n-Lane 50% subkey ranks

Figure 5.17: Evict+time n-Lane percentages

5.7.3. C.ET.3) n-Lane against Evict+Time, reconfiguration interval

In all other experiments in this section the reconfiguration is done for a full encryption. Excluding the noise caused by the reconfiguration cycles itself from the measurements in the process. Although the results in previous experiments are very optimistic there is a huge vulnerability. If the attacker is aware of the Countermeasure, then it should be easy to filter the samples. If the attacker knows we reconfigure for 10% of the iterations of AES, then they can try to reject the 5% samples with longest execution time and 5% shortest execution time and the influence of the Countermeasure is greatly reduced, if not nullified.

A more resistant implementation would be to initiate this random configuration at any point during encryption, so samples where the Countermeasure activates are more difficult to filter. We will test the effect of a more fine grained reconfiguration pattern. We do reconfiguration with a probability of 20%, but we implement this to reconfigure at the start of a random AES round, and also reconfigure back at the end of a random round. On average, half of the rounds will execute in the other configuration, and thus a comparable time is spend in different configurations when compared to when 10% of the encryptions is fully ran in a different configuration. Figure 5.18 shows the comparison between the measurements results of these two different implementations of n-Lane.



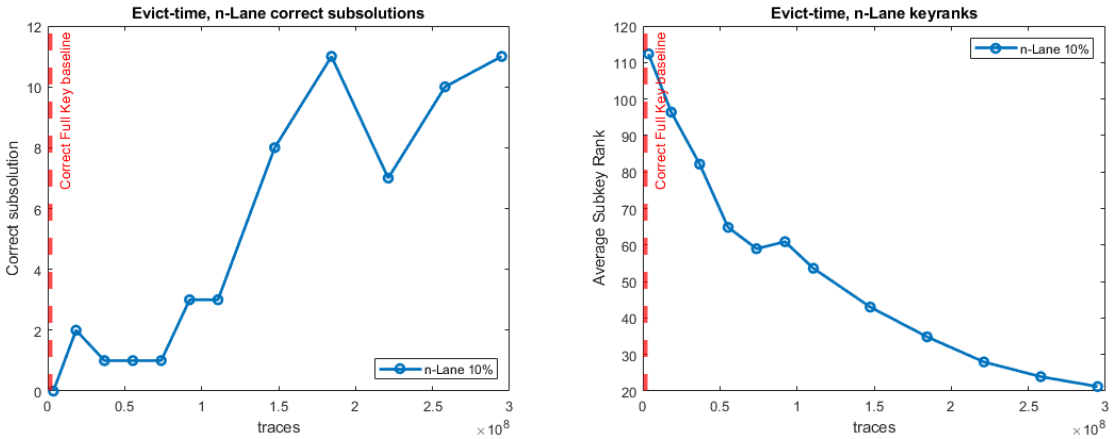
((a)) Evict+time n-Lane 20% between AES rounds, subsolutions ((b)) Evict+time n-Lane 20% between AES rounds, subkey ranks

Figure 5.18: Evict+time n-Lane 20% between AES rounds

We see that the order of interference with the attack is comparable to reconfiguring for a full encryption, and even seems to require a bit more samples to make the attack successful, likely caused by the fact the reconfiguration cycles are now included in the measurements. We thus conclude that the further results we find in the experiments with the n-Lane Countermeasure should be representative even if the implementation is done differently such that the reconfigurations are spread out differently.

5.7.4. C.ET.4) n-Lane against Evict+Time, trace size requirement

We do an additional experiment with the 10% n-Lane Countermeasure with the default reconfiguration pattern. We run for a larger sample size to estimate how many traces we would need to make the attack succefull. Figure 5.19 shows the results for the Evict+Time attack on $3 * 10^8$ timed encryptions on the n-Lane 10% protected AES are presented. The amount of traces required increased with a factor of at least 800 when compared to the unprotected baseline in 5.5.



((a)) Evict+time n-Lane 10% subsolutions ((b)) Evict+time n-Lane 10% subkey ranks

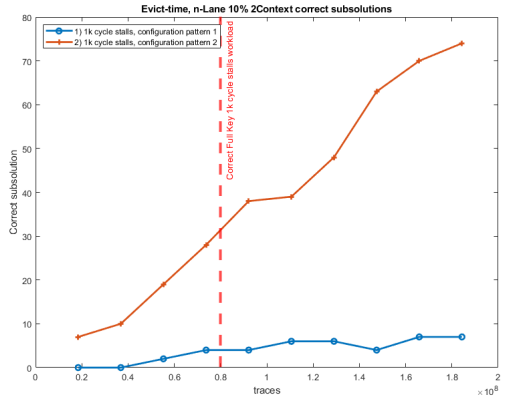
Figure 5.19: Evict+time n-Lane 10%

5.7.5. C.ET.5) n-Lane against Evict+Time, shared processor

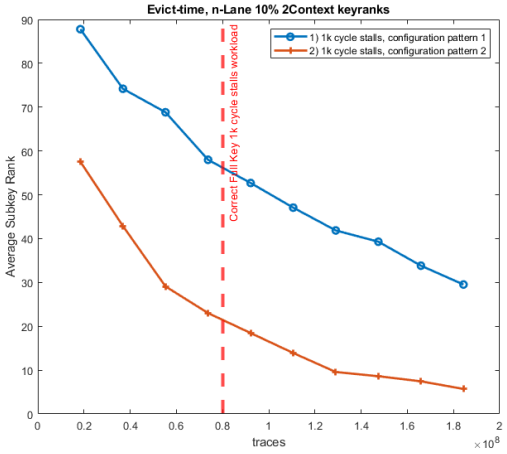
Figure 5.20 shows the results for the combination of sharing the processor with another context, as well as running the default 10% n-Lane Countermeasure. This was tested for both configurations patterns possible to verify which one generated the most noise in this different code setup.

Table 5.7 shows the performance costs on AES throughput of the Countermeasure in this setup, when either compared to the parallel setup on this same workload, like tested in Section 5.6.1, or when compared to the baseline in Section 5.5.1.

We seen an increase of the amount of cycles required for the successful attack when compared to only sharing the system without the Countermeasure active. However, the amount of influence the Countermeasure has on this situation seems to match that without a shared processor. These noise sources do thus not amplify each other, but both get averaged out at a sufficient large sample size, which in this case is larger for the n-Lane Countermeasure.



((a)) Evict+time n-Lane 10% shared processor subsolutions



((b)) Evict+time n-Lane 10% shared processor subkey ranks

Figure 5.20: Evict+time n-Lane 10% shared processor

Evict+Time n-Lane	1) to 2Context	2) to 2Context	1) to baseline	2) to baseline
% average execution time AES	+11.7%	+5.9%	+15.5%	+9.5%

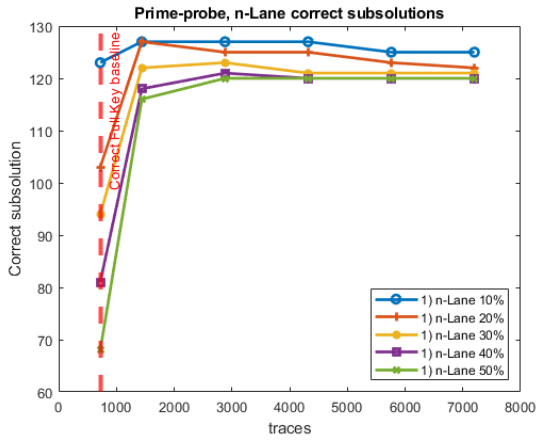
Table 5.7: Overhead costs of Countermeasure on AES performance, in comparison to the performance in either the parallel setup, or the baseline setup.

Evict+Time n-Lane	1) config1	2) config2
% execution time benchmarks	+9.9%	+5.0%

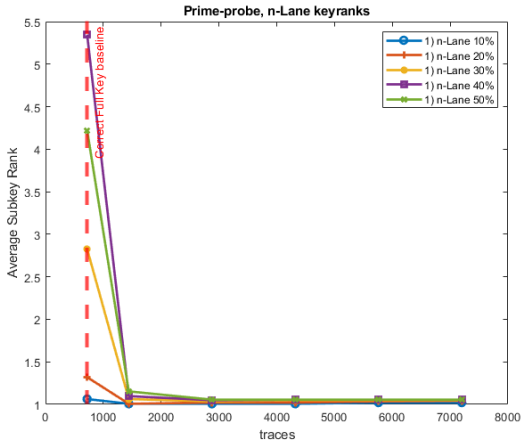
Table 5.8: Overhead costs of Countermeasure on the workload of the second context.

5.7.6. C.PP.1) n-Lane against Prime+Probe

Figure 5.21 shows the results of the default 10% n-Lane Countermeasure implementation against the Prime+Probe attack. Although the n-Lane Countermeasure is not intended to specifically target Prime+Probe attacks, as it is not a timing based attack, the Countermeasure has a small influence on the effectiveness of the attack. In our setup, where we only required 5 samples of plaintext value and keybyte combinations, we now require 3 times as many to get to our maximum amount of correct solutions for any of the percentages chosen. The error on our baseline sample count however is rather insignificant. For a small amount of samples we were also unable to find the correct solution. Certain specific measurements might fail due to the different internal evictions associated with the configurations, making it so a smaller configurations always access a specific Cache line resulting in our attack concluding the wrong most accessed line. This could become more significant if a lot of extra Cache usage is done in the process of calling the AES function, but it is kept to a minimum amount of measurements this affects because of our minimal setup.



((a)) Prime+Probe n-Lane correct subsolutions

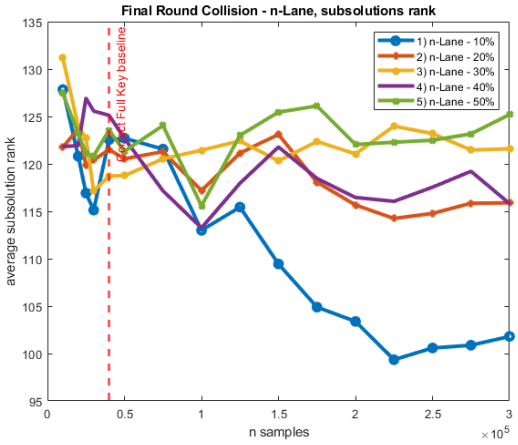


((b)) Prime+Probe n-Lane subsolutions ranks

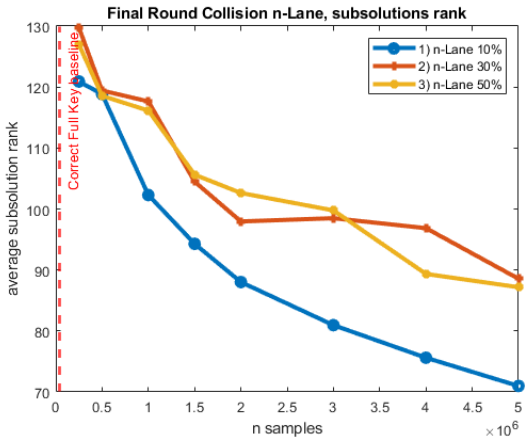
Figure 5.21: C.PP.1) n-Lane against Prime+Probe

5.7.7. C.FR.1) n-Lane against Final Round Collision, reconfiguration percentages

Figure 5.22(a) shows the results of small experiments with different percentage settings of n-Lane, and seem to suggest a similar pattern as the Countermeasure against the Evict+Time attack in Section 5.7.2. Figure 5.22(b) shows the results of a larger sample size of $5 * 10^6$ encryptions for 10%, 30% and 50% n-Lane. We see a smaller influence on increasing the n-Lane percentage than we did with Evict+Time in Experiment C.ET.2), Section 5.7.2. This is likely explained by the fact that the logical behaviour that is used in the attack, the collisions, is not disturbed by the Countermeasure, unlike with the Evict+Time attack.



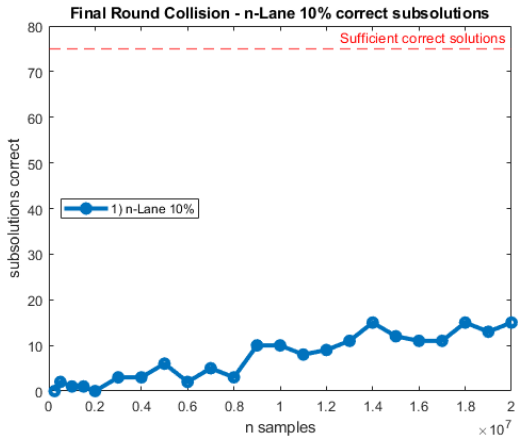
((a)) Final Round Collision n-Lane percentages subkey ranks



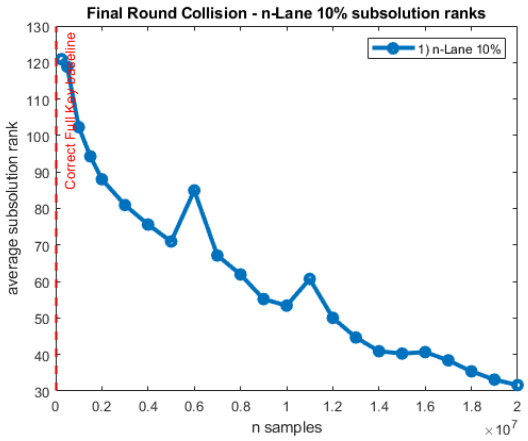
((b)) Final Round Collision n-Lane 10% subkey ranks

Figure 5.22: Final Round Collision n-Lane percentages

Figure 5.23 shows the keyrank for higher sample counts, for the default 10% n-Lane Countermeasure. We estimate at least $3 * 10^7$ samples to be required for a successful attack, and thus at least a factor of 750 more traces, comparable to the factor concluded for the Evict+Time attack in Section 5.7.4.



((a)) Final Round Collision n-Lane 10% subkey subsolutions

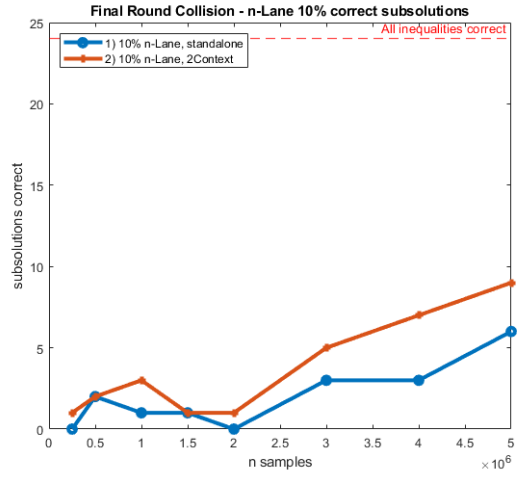


((b)) Final Round Collision n-Lane 10% subkey ranks

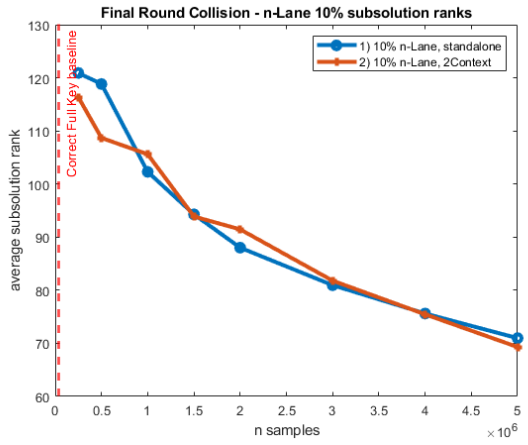
Figure 5.23: Final Round Collision n-Lane percentages

5.7.8. C.FR.2) n-Lane against Final Round Collision, shared processor

Figure 5.24 shows the results of the default 10% n-Lane Countermeasure against the Final Round Collision attack in a shared processor. We see the similar phenomenon as with the n-Lane Countermeasure on top of parallel setup against the Evict+Time attack in experiment C.ET.5) in Section 5.7.5, where is noise of the two noise sources doesn't amplify each other, but only set a requirement for minimal amount of traces to overcome the strongest source. In this case this is the noise caused by the n-Lane Countermeasure.



((a)) Final Round Collision n-Lane 10% with shared processor, subsolutions



((b)) Final Round Collision n-Lane 10% with shared processor, subkey ranks

Figure 5.24: Final Round Collision n-Lane 10% with shared processor

5.7.9. Summary and Discussion

Influence on attack effectiveness

When the AES was protected by the default 10% n-Lane implementation, the Evict+Time Attack was estimated to take over 800x as many samples to be successful. The Final Round Collision attack is estimated to be successful after at least 750x as many samples. We saw a small influence on the effectiveness of the Prime+Probe attack, requiring around 3x as many samples for all correct subsolutions, and making a couple subsolutions immeasurable. For both the Evict+Time and Final Round Collision attacks, we saw no further increase in the amount of samples required when this Countermeasure was applied on top of running on a shared processor.

Overhead costs
The default 10% n-Lane Countermeasure added 8.2% more average execution cycles to the AES encryptions. The overhead scaled roughly linear with the probability to reconfigure. The measured performance cost for workload sharing the processor with the context that encrypts with this Countermeasure, was an average increased execution time of 11.7% for the workload.

Countermeasure implementation details

- Reconfiguring to a smaller configuration had the additional benefit of causing random Cache misses and invalidating samples of the Evict+Time attack. We saw that one configuration generated more noise than the other, and also doubled the overhead of the Countermeasure.
- Balancing between configuring to larger and smaller configurations is possible, keeping the same throughput of AES, while still contributing considerable levels of noise against the attacks.
- Noise caused by sharing the processor had no benefit on top of the n-Lane Countermeasure, as the noise generated by this noise was dominating.
- The implementation used in these experiments is considered representative to different implementations of this Countermeasure. These implementations could divide the time spent in the unique configurations differently, compared to just a full encryption in our implementation. This is shown by testing a different implementation of the Countermeasure.

Discussion
We have seen that the earlier discussed issue of losing cached data between configurations didn't cause the high delay cost that we theoretically associated with this issue as we speculated in Section 4.3. Our overhead cost for 10% n-Lane doubled from 4.1% to 8.2%, what we still consider reasonable. The main reason is that aside from encryptions, the process had little extra operations, allowing data that got cached twice in the 4-way Cache due to it missing in the smaller configuration to remain as a copy in the Cache. This might however become an issue if there is more Cache interference in between encryptions.

The option to briefly assign more execution lanes to the encrypting context to generate noise, and thus briefly increase throughput of encryptions, is a upside in system setups where we let the ρ-VEX operate on scheduled tasks with different priorities. We saw reconfiguration patterns where we only increased the average encryption time with 4.6% and 1.2% while considerably increasing the amount of traces required when compared to our baseline attack.

For more disruptive patterns with high percentages and even with 50% chance to reconfigure the overhead cost stayed below the minimal cost of disabling the Cache in two rounds, 41% overhead when compared to at least 49% overhead for Cache bypass of the table lookups in two rounds.

Two downsides associated with this Countermeasure is the fact that this Countermeasure is mainly purely timing noise, meaning that if our attack already requires a lot of samples then we do not gain much benefit from this Countermeasure. Another is that it is relatively easily detectable if the system is relatively noiseless aside from this Countermeasure if implemented like we did. Samples executed in the different configurations can then be clustered based on execution time.

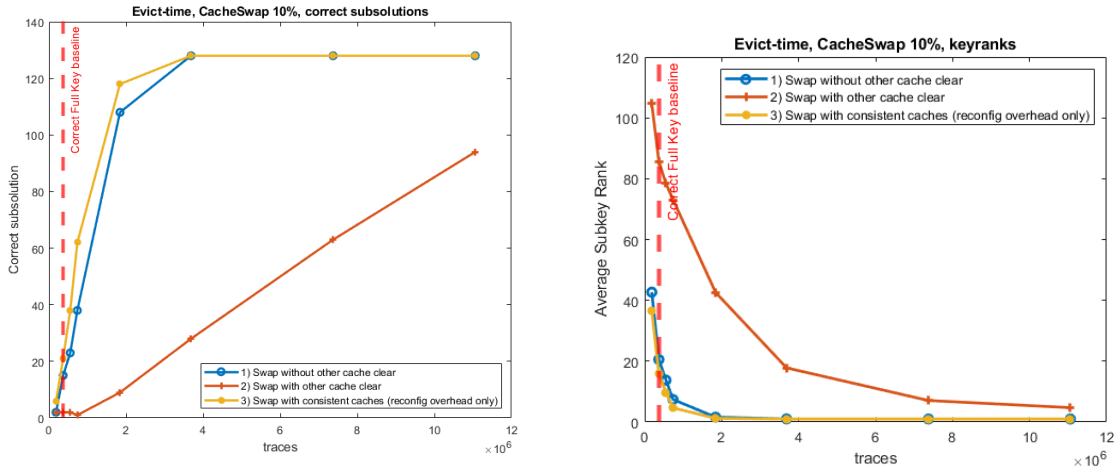
5.8. Experiment Set D: Cache Attacks against CacheSwap protected AES

5.8.1. D.ET.1) CacheSwap against Evict+Time, Cache state

Figure 5.25 shows the results of experiments on the CacheSwap Countermeasure, where the influence of the state of the secondary Caches on the processor, not used by the encrypting process is tested. The experiments are done on the CacheSwap Countermeasure with 10% swap chance, in three different scenarios. Table 5.9 shows the measured performance cost in those scenarios. In experiment 1) the attack runs in isolation, the Cache state of the other system Caches is not touched and will eventually contain all AES related data. The only effect we thus have from the Countermeasure is the overhead of reconfiguration cycles, and the sample occasionally not containing the correct information because not all copies of the data are evicted, only that in the main Cache. In experiment 2) we fully clear the other Caches after every attacked encryption. This emulates what happens if other system resources do

evictions to these random memory lookups that are redirected to the other Cache, and a reconfiguration thus also adds noise in form of Cache misses. In experiment 3, we guarantee a consistent Cache state between the two Caches that we swap between, by doing both the setup encryption and the Cache eviction in both Caches. This way we measure the noise from the reconfiguration overhead only. We see that the random Cache misses that occur during these brief swaps add considerable more noise than the logical effect that certain samples become invalid due to the reconfiguration pattern in either the setup or measured encryption. This can be seen by the increase of average keyrank from experiment 1 to 2, but also by the fact that experiment 1 and 3 have very similar performance. We will test this attack on multiple percentages of CacheSwap against Evict+Time, but to see how the logical behaviour changes with the different percentages the measurements against the Prime+Probe attack in Experiment D.PP.1), Section 5.8.5, should be a better indicator. We conclude that for this ideal experiment 2, 10% CacheSwap where the other Cache is cleared, the amount of traces required increases with a factor estimated at 40 for a average execution time increase of 9.5%.

We will continue with emulating a different process by clearing the second Cache between encryptions in the standalone experiments for CacheSwap against all attacks in the rest of the experiments. In the experiments on the shared processor this ideal clearing is replaced with a second context doing Cache evictions because of its own lookups.



((a)) Evict+time CacheSwap 10% correct subsolutions

((b)) Evict+time CacheSwap 10% subkey ranks

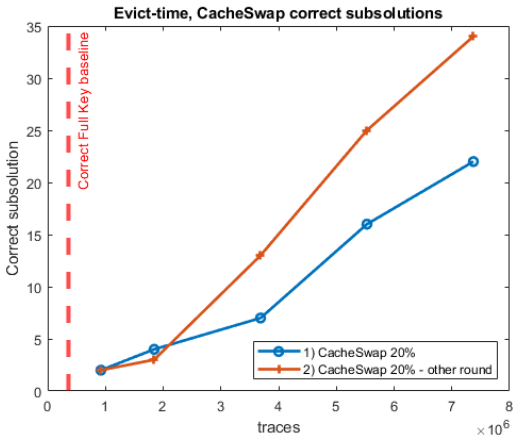
Figure 5.25: Evict+time CacheSwap 10%

Evict+Time CacheSwap 10%	1) no clear	2) clear	3) consistent Cache
% average execution time AES	+5.4%	+9.5%	+5.4%

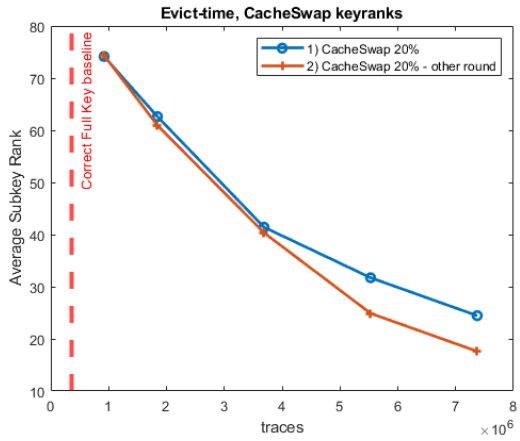
Table 5.9: Overhead costs of the 10% CacheSwap Countermeasure, based on the Cache state of the second Cache.

5.8.2. D.ET.2) CacheSwap against Evict+Time, targeted round effect

Figure 5.26 shows the results of CacheSwap 20% against the Evict+Time attack. In experiment 1 we implement it as the default, but in experiment 2 we protect the second AES round instead of the first round. We thus set the difference between the scenarios like in Experiment D.ET.1), Section 5.8.1, where we effectively measure how much the invalidated samples contribute to the noise on the attack. We see an apparent larger difference between the two scenarios than with 10% CacheSwap for this higher percentage 20%. Meaning the logical influence on the attack by this reconfiguration pattern starts becoming more significant for this percentage.



((a)) Evict+time CacheSwap 10% correct subsolutions



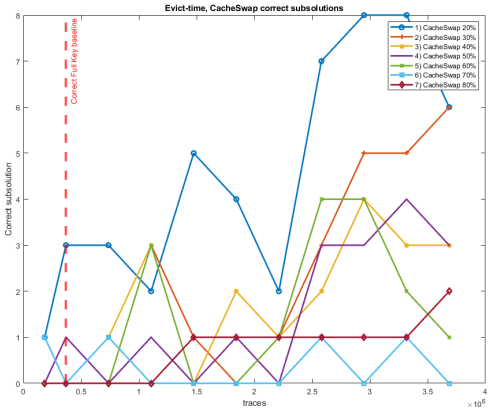
((b)) Evict+time CacheSwap 20% subkey ranks

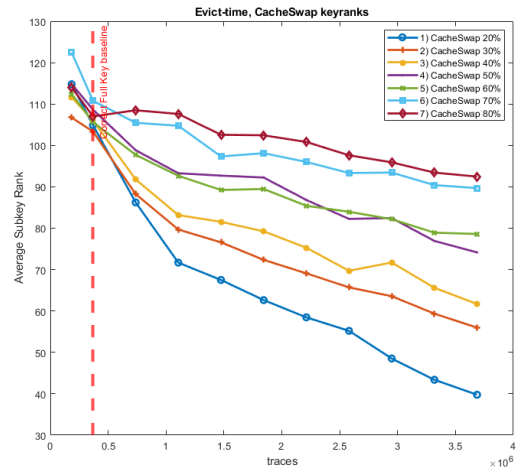
Figure 5.26: Evict+time CacheSwap 20%

5.8.3. D.ET.3) CacheSwap against Evict+Time, reconfiguration percentages

Figure 5.27 shows the effect of the reconfiguration chance of the CacheSwap Countermeasure on the performance of the Evict+Time attack in an isolated setup with a empty second Cache. Table 5.10.

We see the overhead decrease for percentages above 60%. This can be attributed to the fact that now more lookups between setup encryption and timed encryption will result to hits in the second Cache, but also because effectively less configurations will be issued since the CacheSwap swaps will switch Caches less often. This is partially caused by our setup: if a heavy workload runs in parallel in the second Cache, than the overhead costs can increase further beyond this 60%, as we do not clear between our setup encryption and measured encryption in this setup.





((a)) Evict+time CacheSwap percentages correct subsolutions ((b)) Evict+time CacheSwap percentages subkey ranks

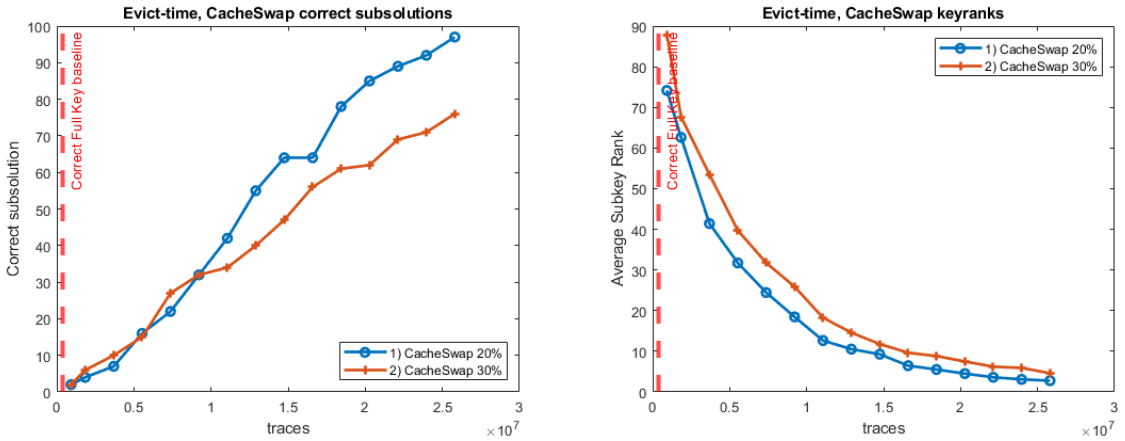
Figure 5.27: Evict+time CacheSwap percentages

Evict+Time CacheSwap percentages	1) 20%	2) 30%	3) 40%	4) 50%	5) 60%	6) 70%	7) 80%
% average execution time AES	+14.6%	+18.7%	+21.7%	+23.5%	+24.0%	+23.3%	+21.9%

Table 5.10: Overhead costs of different reconfiguration percentages of the CacheSwap Countermeasure, second Cache cleared between samples

We repeat the experiment with 20% and 30% CacheSwap with a higher sample count, to get a

indicator for how much extra traces are required for higher percentages. we see that even when we nearly have a factor of 70 more samples, not all subsolutions show the correct result. Following the current pattern, both should be successful around a factor 100 more samples. We also see that the difference between 20% and 30% is minimal, but the difference with experiment with 10% CacheSwap in Section 5.8.1 is more significant.

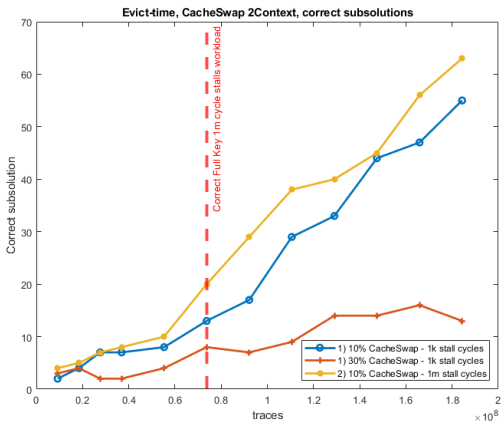


((a)) Evict+time CacheSwap percentages correct subsolutions ((b)) Evict+time CacheSwap percentages subkey ranks

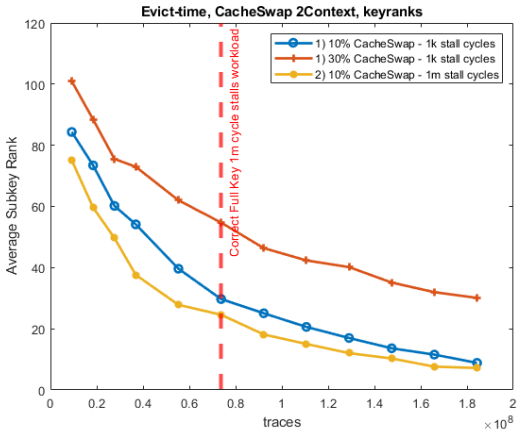
Figure 5.28: Evict+time CacheSwap percentages

5.8.4. D.ET.4) CacheSwap against Evict+Time, shared processor

Figure 5.29 shows the results of experiments of the CacheSwap Countermeasure against the Evict+Time attack running on a shared processor setup. Table 5.11 shows the increase of average AES executon time, and the increased execution time of the benchmarks due to this pattern. We measure 13.3% and 15.9% overhead for the Countermeasure for 1m and 1k stall cycle benchmark workloads respectively, as opposed to the 9.5% measured in the baseline with a cleared Cache in Experiment D.ET.1), Section 5.8.1, this because now also the state of the encryption Cache is disturbed by the the parallel context, and we no longer keep our instruction Cache data consistent in the other resources between encryptions. The measured overhead for the context executing benchmarks is 8.16% and 8.94% for CacheSwap 10% on the 1m and 1k stall cycle benchmark workloads respectively, and for CacheSwap 30% on 1k stall workload it is 13.66%. With the shared processor baseline measurements in Experiment B.ET.1), Section 5.6.1, we found for a 1m workload the attack required around 100 times as many samples for a successful attack, and for the 1k workload 200 times as many samples. For 10% CacheSwap this increases to around a factor 800 for the high 1k workload, and a bit under that for low 1m workload. For 30% we are not able to estimate how many more samples are required, but the difference in factors when compared to 10% seems to be increased than with the previous measurements on the isolated case.



((a)) Evict+time CacheSwap 10% shared processor correct subsolutions



((b)) Evict+time CacheSwap 10% shared processor subkey ranks

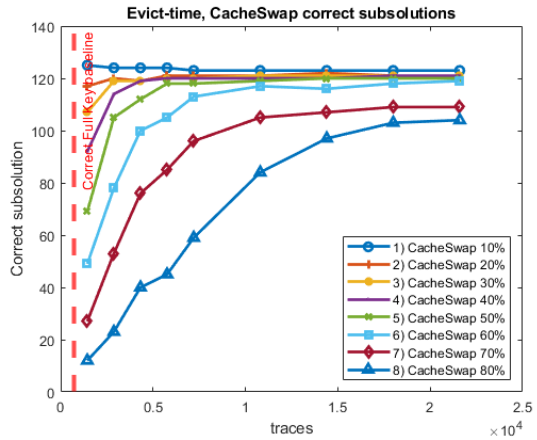
Figure 5.29: Evict+time CacheSwap 10% shared processor

E+T CacheSwap Shared Processor	1) 10%, 1m workload	2) 10%, 1k workload	3) 30%, 1k workload
% average execution time AES	+11.5%	+11.8%	+14.4%
% average benchmark execution time	+8.16%	+8.94%	+13.66%

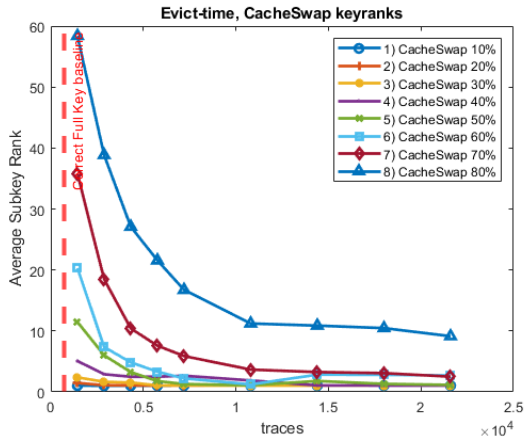
Table 5.11: Performance cost caused by CacheSwap on the parallel context in a shared processor setup

5.8.5. D.PP.1) CacheSwap against Prime+Probe, reconfiguration percentages

Figure 5.30 shows the effect of the reconfiguration chance of the CacheSwap Countermeasure on the performance of the Prime+Probe attack in an isolated setup with a empty second Cache. For the lower percentages there is barely an effect on the effectiveness of the attack. Even for an high percentage of 50%, it seems that only around a factor of 7 times more traces are required to reach a level where nearly all subsolutions are found. But even before that point, when there are only twice as many traces used as for the minimal amount of traces in the baseline attack, already 70 subsolutions are found so there is high probability that the key is already fully retrievable in offline processing step.



((a)) Prime+Probe CacheSwap percentages correct subsolutions

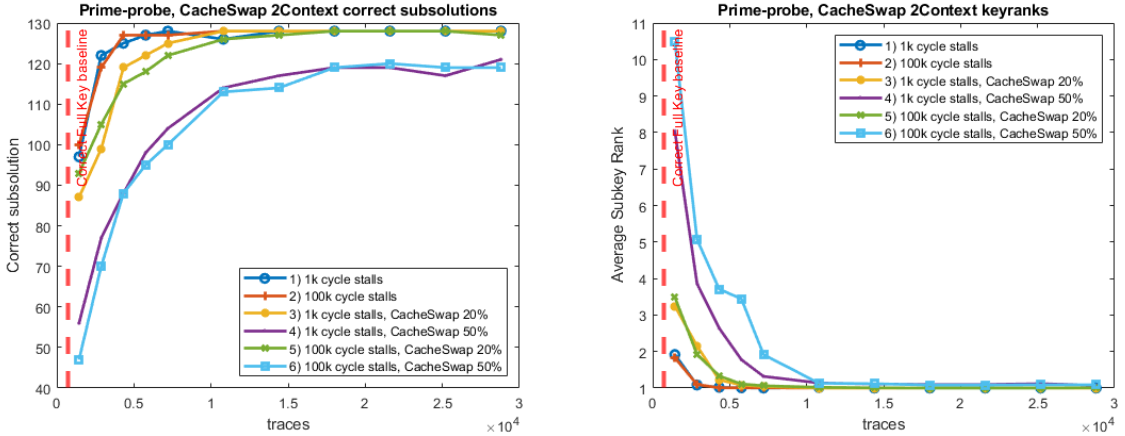


((b)) Prime+Probe CacheSwap percentages subkey ranks

Figure 5.30: Prime+Probe CacheSwap percentages

5.8.6. D.PP.2) CacheSwap against Prime+Probe, shared processor

Figure 5.31 shows the effect of the reconfiguration chance of the CacheSwap Countermeasure on the performance of the Prime+Probe attack in an isolated setup. We need roughly a factor of 12 more samples before we get all correct samples for 20% CacheSwap on the workloads, but this relatively close to the baseline shared setup without CacheSwap. For CacheSwap 50% however, a factor 25 total, or roughly a factor 3 of extra samples when compared to the shared processor setup was required. The average keyranks already start relatively low for a lower amount of samples, like in the isolated setup in Experiment A.PP.1), Section 5.5.2, and multiple correct subsolutions are already found.

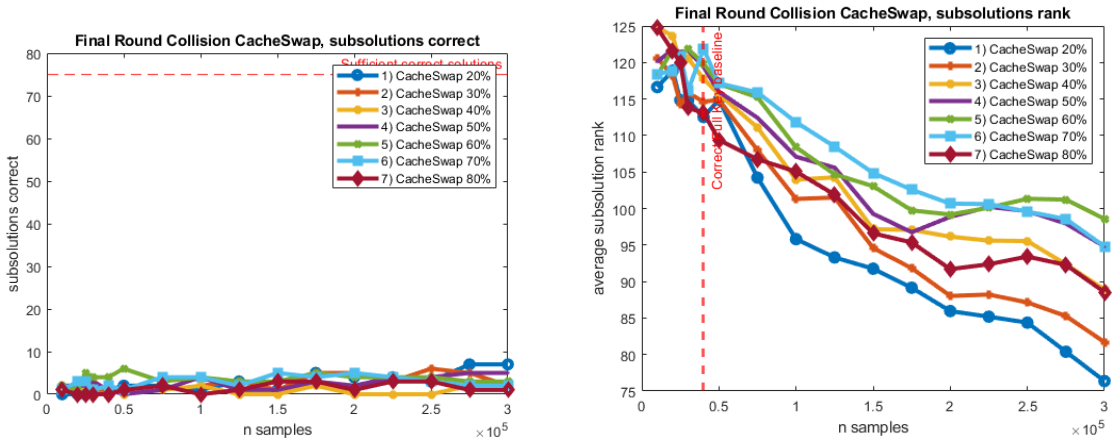


((a)) Prime+Probe CacheSwap shared processor correct sub- ((b)) Prime+Probe CacheSwap shared processor subkey ranks solutions

Figure 5.31: Prime+Probe CacheSwap shared processor

5.8.7. D.FR.1) CacheSwap against Final Round Collision, reconfiguration percentages

Figure 5.32 shows the results of the Final Round Collisions attack against AES protected with the CacheSwap Countermeasure with percentages varying from 20% to 80%. For this small sample sizes we do not have enough information yet on the increase of sample size requirement, but the results suggest the influence of the Countermeasure is stronger than on the previous two attacks. It is also shown that, with CacheSwap with percentages higher than 50%, more collisions happen again in the other Cache. We thus see average key ranks decrease for higher percentages. For instance, for 70% CacheSwap, the same amount of collisions as 30% CacheSwap is expected. 70% CacheSwap does seem to cause more noise than 30% in this experiment, likely due to more noise caused by the reconfiguration overhead.



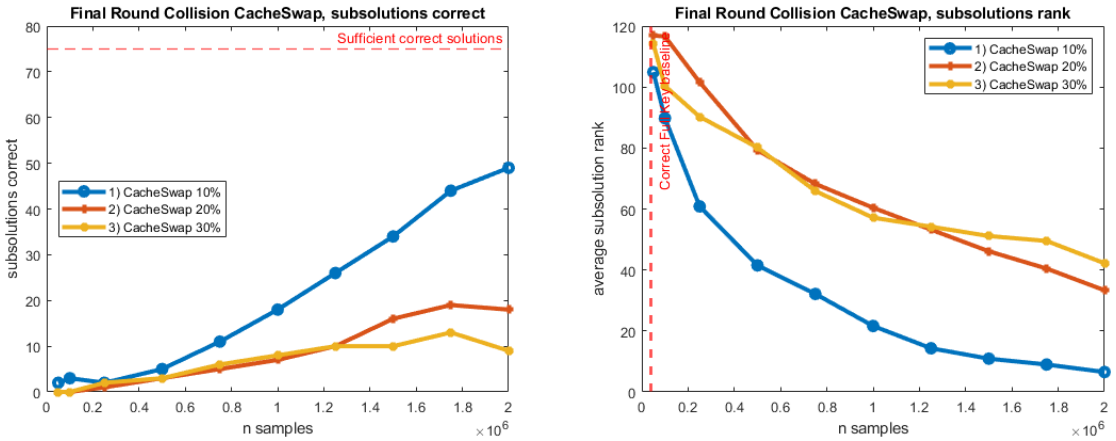
((a)) Final Round Collision CacheSwap percentages correct subsolutions

((b)) Final Round Collision CacheSwap percentages subsolution ranks

Figure 5.32: Final Round Collision CacheSwap percentages

‘

We test the percentages 10% to 30% on a higher samples size of $2 * 10^6$ encryptions. Figure 5.33 shows the results of these experiments. We see a similar behaviour of the Countermeasure when increasing the chance to swap. Where the jump from 10% to 20% makes a larger difference than the jump from 20% to 30%. We estimate the attack to take 50 times as many traces to be successful when AES is protected with 10% CacheSwap.



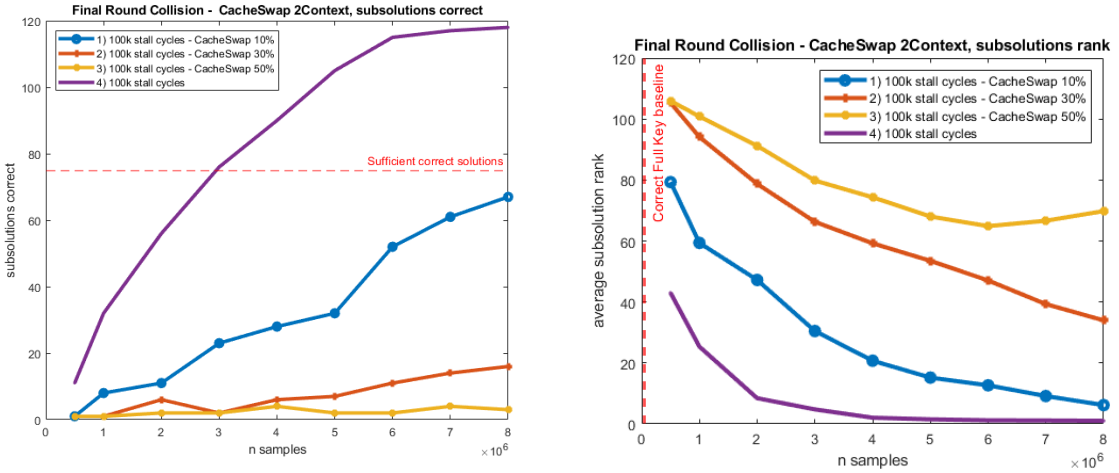
((a)) Final Round Collision CacheSwap percentages correct subsolutions

((b)) Final Round Collision CacheSwap percentages subkey ranks

Figure 5.33: Final Round Collision CacheSwap percentages

5.8.8. D.FR.2) CacheSwap against Final Round Collision, shared processor

Figure 5.34 shows the result of the CacheSwap Countermeasure against the attacked encryption process running in shared processor, with the PowerStone benchmark workload with 10^5 stall cycles between benchmarks. The baseline performance of the attack against AES sharing the processor with this workload is also included, as this workload was not tested in Experiment B.FR.1), Section 5.6.3. The amount of samples required for a successful attack when 10% CacheSwap is added, increases with a factor of roughly 3 when compared to the shared processor setup with this workload, with a factor 8.5 compared to the isolated CacheSwap setup and a factor 225 when compared to the unprotected baseline.



((a)) Final Round Collision CacheSwap shared processor correct subsolutions

((b)) Final Round Collision CacheSwap shared processor subsolution ranks

Figure 5.34: Final Round Collision CacheSwap shared processor

5.8.9. Summary and Discussion

Influence on attack effectiveness

When the AES encryption was protected by the default 10% CacheSwap implementation, the Evict+Time Attack was estimated to need 40x as many traces to succeed. The Prime+Probe Attack was seemingly unaffected by lower percentages up to 20% of CacheSwap in the isolated setup, and only increasing with a factor 10x for the higher CacheSwap percentage 50%. The Final Round Collision Attack was estimated to take 50x as many traces when protected under 10% CacheSwap. In the shared processor setup, the trace amount required for the Evict+Time attack increased to 800x, this was 4x as many as the unprotected shared processor setup. For a lower workload, the effect seemed similar and thus the noise was increased more by introducing the Countermeasure. The Prime+Probe attack became more noticeably affected by this Countermeasure in the shared processor setup, this was still only significant for higher swap percentages, where 50% CacheSwap on top of a high workload increased the amount of samples required for a successful attack from a factor 8 to a factor 25. For the Final Round Collision attack in this setup 225x as many traces were required, 3x as many as the parallel processing baseline.

Overhead costs

The measured overhead cost of the 10% CacheSwap Countermeasure, without extra overhead costs from evictions caused by a process briefly swapped to the main context, was 9.5% in the worst case of the other Cache being fully cleared. This increased to up to 24.0% for 60% CacheSwap, decreasing beyond this percentage. For a shared processor setup with the highest workload, the overhead for AES throughput increasing from 9.5% to 11.5% for 10% CacheSwap, and saw the overhead costs increase less for increased percentages CacheSwap than with our isolated setup. The average execution time of the workload sharing the processor increased by 8.94% for 10% CacheSwap.

Countermeasure implementation details

- The access based noise seems to only become noticeable for higher percentages and when combined with other contexts that run workloads that cause evictions during swaps.
- The noise caused by the combination of CacheSwap and sharing the processor amplify each other, gaining benefits when both are combined.
- The diminished returns on CacheSwap above 50% against Collision Attacks seems to become apparent above 60%, still gaining noise by the reconfiguration pattern above 50% CacheSwap.

Discussion

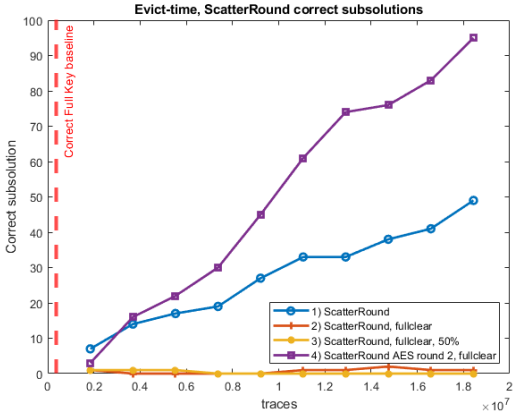
In the isolated setup, the timing noise caused by CacheSwap was much weaker when compared to the n-Lane Countermeasure tested in the previous section, while the cost associated with it was higher. It is

only when combined with the shared processor setup that it can become the same order of significance as the n-Lane Countermeasure. We saw the combination of a high workload, 1k stall cycles, and 10% CacheSwap against Evict+Time reach the same order of trace count demand, 800x. A lower parallel workload, 100k stall cycles, in combination with 10% CacheSwap on the Final Round Collision Attack did not reach the same level as the n-Lane Countermeasure did against this attack. For CacheSwap above 50% the overhead cost became more than bypassing the Cache on the table lookups for a full round, making it undesirable to go up to this percentage if the attacker could likely just bypass the Cache for that round. Aside from diminishing returns on Collision attacks, the attacker has to also question if for these higher percentages, they would rather just look at mechanisms to fully isolate this round from the attacker process. Although the logical noise on the current implementation of the ρ-VEX seemed to be very insignificant, even for higher percentages, it can become way more significant if the Cache was implemented with larger Cache lines. Using the most common Cache line length, $\delta = 16$, then there is a much higher chance a briefly swapped process does evictions to the AES table data. However, with this Cache line size, the penalty of a Cache miss also increase and thus our measurements in this thesis are no longer indicative of performance cost. Updating the ρ-VEX design to include larger Cache lines by itself can be beneficial to both performance as well as resistance against Cache attacks, but could also make this Countermeasure significant enough to consider implementing.

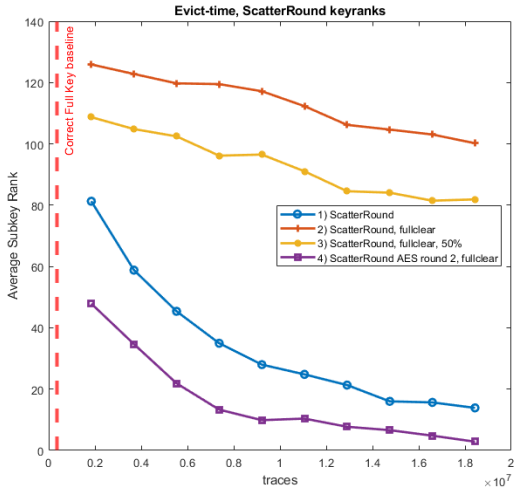
5.9. Experiment Set E: Cache Attacks against ScatterRound protected AES

5.9.1. E.ET.1) ScatterRound against Evict+Time

Figure 5.35 shows the results of the ScatterRound Countermeasure against the Evict+Time attack, tested in four scenarios. In Experiment 1 we only run the reconfiguration of ScatterRound, and we clear the Cache of the main AES context, so the effect on the setup process becomes relevant. In Experiment 2 we also clear the Cache that is not assigned to the encrypting context, so switching to those Caches will cause Cache misses, and more accurately allows to estimate the performance cost. Experiment 3 repeats Experiment 2, but with only 50% of the encryptions are scattered. Experiment 4 applies the same scattering condition as in Experiment 2, but now the second round is scattered instead of our targeted first round.



((a)) Evict+time ScatterRound correct subsolutions



((b)) Evict+time ScatterRound subkey ranks

Figure 5.35: Evict+time ScatterRound

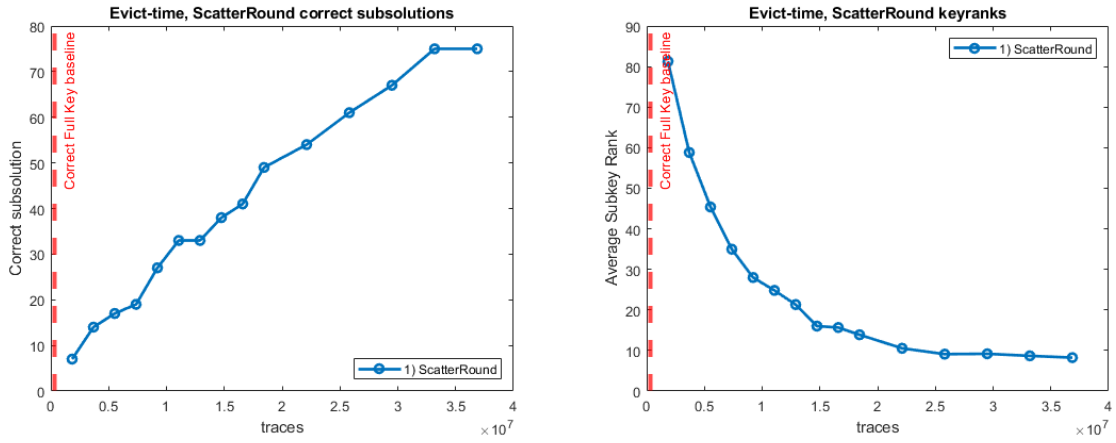
Evict+Time ScatterRound	1) 100% full clear	2) 50% full clear
% average execution time AES	+63.2%	+31.7%

Table 5.12: overhead costs of ScatterRound Countermeasure

Table 5.12 shows the overhead cost associated with the ScatterRound Countermeasure, measured in experiments 2 and 3, with a cleared Cache. This is when the main configuration is a 4-way Cache. The reconfiguration overhead, decreased instruction throughput in the smaller configuration, Cache misses in the secondary Caches and interference on the own Cache state all contribute to high overhead costs for this Countermeasure. ScatterRound applied to only one round can require more performance cost than Cache bypassing the lookups of two full rounds, as shows in Table 5.3 in Section 5.2

We see that the noise coming from the logical influence on invalidating samples is much higher than the noise coming from the reconfiguration pattern. The logic behaviour is close to CacheSwap with higher reconfiguration probabilities. The timing noise is also less, since there is a constant amount of reconfigurations. This is also why ScatterRound with 50% scatter chance has a relatively high influence on the attack, since this adds more timing noise through randomly distributed reconfigurations.

We repeat Experiment 2 on a larger sample size to estimate how many samples it would require to successfully perform the Evict+Time attack. Figure 5.36 shows the results of this experiment. We estimate the amount of samples required for the attack to be successful to increase with at least a factor 160.



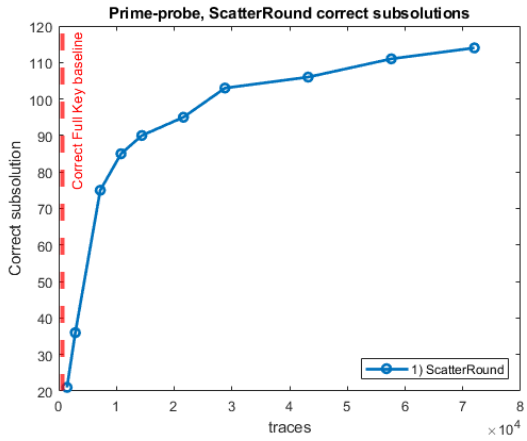
((a)) Evict+time ScatterRound correct subsolutions

((b)) Evict+time ScatterRound subkey ranks

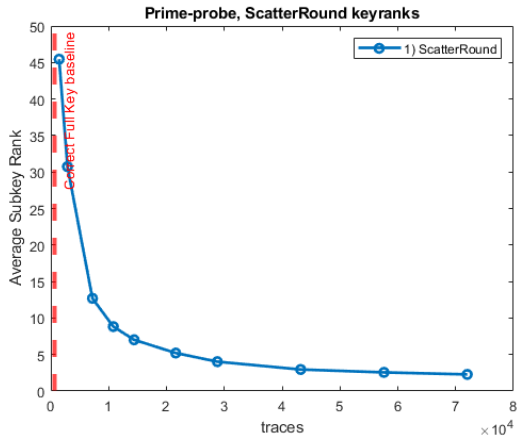
Figure 5.36: Evict+time ScatterRound

5.9.2. E.PP.1) ScatterRound against Prime+Probe

Figure 5.37 shows the results of the ScatterRound Countermeasure against the Prime+Probe attack. It is implemented with a full clear of all Cache of the Caches not belonging to to the encrypting context between encryptions. In these measurements, only 1/4th of the samples correctly access the Cache with the data we target. We see this in our results: at a factor 15 more traces we only have half of the correct measurements. At the end, at a factor 175 more traces, we do not have all correct measurements yet.



((a)) Prime+Probe ScatterRound correct subsolutions



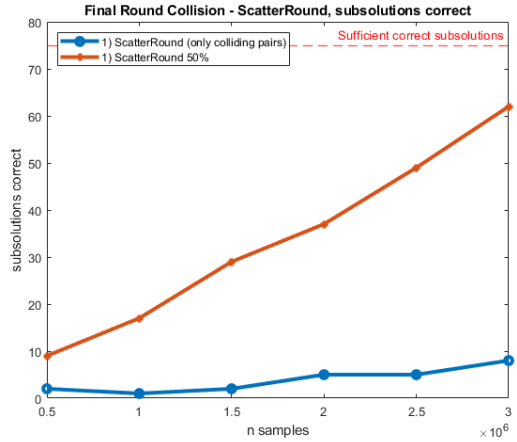
((b)) Prime+Probe ScatterRound subkey ranks

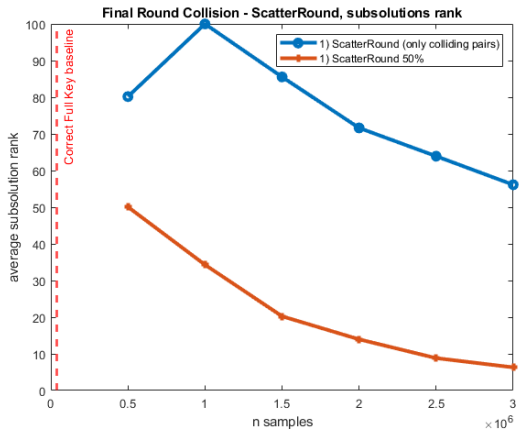
Figure 5.37: Prime+Probe ScatterRound

5.9.3. E.FR.1) ScatterRound against Final Round Collision, standalone execution

Figure 5.38 shows the results of the Final Round Collision attack against the ScatterRound protected AES. We additional test the results if only 50% of the encryptions is protected with the Countermeasure. The average subsolution rank plot in Figure 5.38(a) only shows the average subsolution rank of the keybyte combinations that are allowed to collide under this Countermeasure: the 24 pairs consisting of keybytes in the same column.

We see that, now that 50% of samples do not contain the information we expected them to have, that the amount of samples required to successfully do this attack increases roughly with a factor 100. For the full ScatterRound Countermeasure this sample size was not sufficient to accurately estimate the amount of traces required, but under the current trend it should be atleast above $9 * 10^6$ samples, an increase of a factor 225 over the baseline.





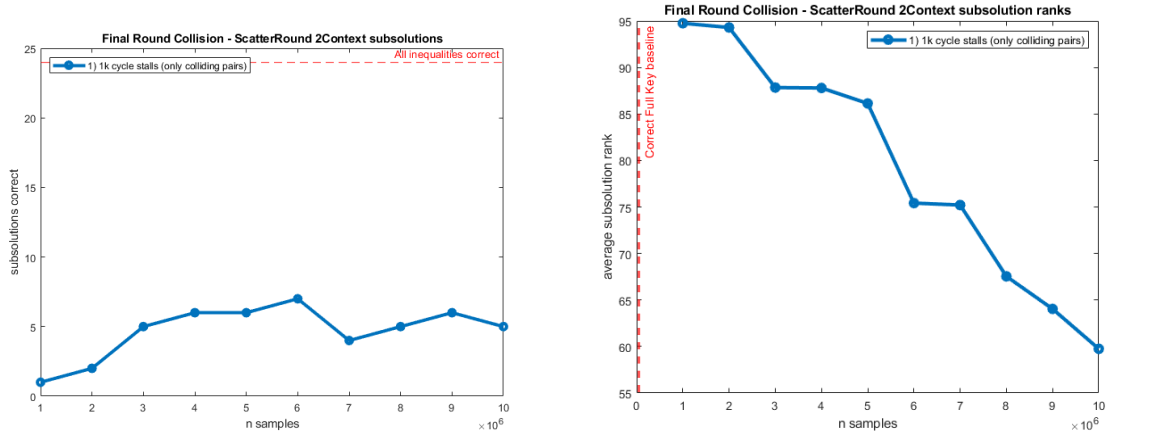
((a)) Final Round Collision ScatterRound patterns subsolution ((b)) Final Round Collision ScatterRound subsolution ranks ranks

Figure 5.38: Final Round Collision ScatterRound

5.9.4. E.FR.2) ScatterRound against Final Round Collision, shared processor

Figure 5.39 shows the combined influence of ScatterRound and sharing the processor. The average subsolution ranks decreases much slower than either of these two noise sources alone. Estimating that we would need well over $2 * 10^7$ samples and thus at least 500 times as many samples as in the

isolated unprotected setup. This is an increase of a factor 2.5 as the amount of samples required for the shared processor with this workload.



((a)) Final Round Collision ScatterRound shared processor subsolution ranks

((b)) Final Round Collision ScatterRound shared processor subsolution ranks

Figure 5.39: Final Round Collision ScatterRound shared processor

5.9.5. E.EX.1) ScatterRound performance in a dedicated execution mode

In Table 5.13 the results of the performance experiments are done. We express the performance losses as an increased percentage of execution time when compared to execution of encryptions in a single context in 8-way mode. We see the performance cost when going from 8-way to 2-way we measure +227% extra cycles execution time, a bit higher than our baseline measurements in Section 5.2, where the increase was from 323 to 912 cycles or +180% extra execution time. This is likely caused by the Cache behaviour that changed for this setup, where more internal evictions (instructions and data) happen in this setup than the baseline experiment.

AES performance	8-way, 1 context	2-way, 1 cntx	2-way, 4 cntx	2-way, 4 cntx, ScatterRound
% execution time AES	+0.0%	+227%	+81%	+99%

Table 5.13: Overhead costs of the reconfiguration percentages in Figure 5.16

We then activate all other contexts, so 4 encryptions are done in parallel. We see the slowdown drop by a factor 2.8 instead of 4, since all contexts can stall each other because of the shared memory bus. Finally, we test this same setup but now with our ScatterRound implementation like discussed in Section 4.5.4, where a context randomly rotates all contexts over the hardware resources in vulnerable round (in this case the first round) for a setup executing 4 AES encryptions in parallel. and can briefly stall other contexts that want to do the same.

Adding in 9.5% extra execution time per encryption when compared to the 4 parallel encryptions without the Countermeasure. The amount of extra cycles measured, 109, is just a bit higher than the computed best case extra execution time per encryption Section 4.5.4 of 96 cycles. This is can be caused by extra reconfiguration delay or because our Countermeasure implementation can stall our contexts for the reconfiguration privilege system.

5.9.6. Summary and Discussion

Influence on attack effectiveness

The increase in amount of encryptions required for a successful Evict+Time attack was estimated to be 160x for the isolated setup. For the Prime+Probe attack, where we did not adjust our attack to keep all Cache space in mind, the amount of samples required increased with factor 175x. The increase for the Final Round Collision was 225x in the isolated setup, and 500x in the shared processor setup for the highest workload, an increase of 2.5x of the attack on the shared processor with just this workload.

Overhead costs
Applying the ScatterRound Countermeasure on a single round of the AES encryptions running on a 4-way ρ-VEX configuration increased the average cycle count per encryption with 63.2%, unacceptable performance wise. For a dedicated setup, with four parallel encryption contexts in 2-way mode, the Countermeasure caused 9.5% performance cost compared to this operation mode without the Countermeasure, but measured to be 99% slower than if the processor was used in a 8-way mode without countermeasures.

Discussion
Aside from being able to fully prevent a collision attack for rounds implemented with the 4 T-Tables, we saw additional benefits. Making collision attacks targeting a S-Box AES round implementation, like our Final Round Collision attack, take more offline encryptions and require at least 225x as many online samples. Against the Evict+Time and Prime+Probe we identified similar benefits of the Countermeasure as achieved by the CacheSwap Countermeasure, and as Collision attacks through the tracing Side-Channel is usually done on the first round, this would advocate for implementing this Countermeasure on both the first and final round.

We concluded that the overhead associated with this Countermeasure is unfeasible when implemented on just an encrypting context running in isolation. However, in our dedicated setup we realised a very reasonable overhead cost of 9.5% when compared to this setup when it is unprotected. This setup however ends up having a total of 99% increased average encryption time than when the full processor is used in 8-way mode. This setup also potentially has further security benefits: four parallel encryptions cause timing noise between each other. The constant rotation between Caches can also complicate access based attacks, as it becomes unclear what Cache to target and potentially requires privileges to access all Cache.

This setup is especially interesting if the ρ-VEX is used as a co-processor in a multi processor system. User access to this processor can be limited because the use of the processor is mainly determined by a scheduling CPU determining what tasks to route to this core. This by itself can make the encryptions done on the ρ-VEX more resistant or potentially to access attacks on its private Cache levels, and makes timing attacks a greater concern, which we have shown the ρ-VEX Countermeasures to be more effective against timing based attacks than against access based attacks.

5.10. Conclusion

In this chapter, we did experiments on Cache Attacks and Countermeasures on a Genesys2 FPGA development board running the default ρ-vex design, with increased Cache capacity to benefit our attacks and Countermeasure performance. The standalone GRLIB version of the ρ-VEX system was ported to this board to use the on-board DDR3 memory chip. The OpenSSL AES implementation with Table T4 in the final round was ported to the system. We ran the attacker process and the AES code in a single process in a single hardware context, defaulting to a configuration that uses 4 of the 8 execution lanes. We made two setups. The first setup was a standalone setup where this process ran in isolation, and effects of a shared processor were partially emulated. In the second setup a second context executing a random selection of PowerStone benchmarks was running, with a configurable amount of stall cycles. This emulates a shared processor with varying workloads.

Executed in isolation, our attacks required comparable amounts of samples to that found in literature, but consistently less because of the smaller Cache lines and noiseless setup. Sharing the system with another workload significantly affected our timing attack efficiency, taking up to 200x as many samples for the Evict+Time and Final Round Collisions Attacks for the highest workload. The access based attack Prime+Probe also took up to 10x as many samples because of the shared memory system.

The n-Lane Countermeasure was mainly tested with a 10% chance to randomly reconfigure between the three configuration sizes at the start of an encryption. The overhead cost was 8.2% in the isolated setup, and increased roughly linear when increasing this percentage. The required amount of samples increased 800x for the Evict+Time attack and 750x for the Final Round Collision attack. Although not intended, the Prime+Probe attack also required 3x as many encryptions for all correct samples, but had no significant influence on the success of the attack. In a shared processor, the overhead cost for the parallel workload was 11.7%. However, running the n-Lane Countermeasure on this setup had no additional benefit when compared to the baseline. We saw that when trying to

optimize the performance, the noise was significantly reduced. The cost of the unoptimized version was considered acceptable.

The CacheSwap Countermeasure was mainly tested with a 10% chance to randomly swap Caches with another process, during the computation of one column of a protected AES round. The overhead cost was 9.5% in the isolated setup , and increased to a maximum of 24% for 60% CacheSwap. The overhead cost increased to 11.5% in a shared processor setup. The required amount of traces increased with 40x for Evict+Time, 50x for Final Round Collision, but Prime+Probe was unaffected for lower percentages of CacheSwap, and only increasing with a factor 8x for the higher CacheSwap percentage 50%. When testing on the shared processor, the noise from the Countermeasure became for significant. 10% CacheSwap caused 8.94% extra execution cycles for the workload. For a high workload with 1k stall cycles, Evict+Time required 800x as many samples as the baseline, 4x as many as the unprotected shared setup. A lower workload with 1m stall cycles the increase was comparable. The Final Round Collision required 225x with a medium workload. The 50% CacheSwap against the Prime+Probe attack made the trace requirement increase from 8x to 25x as many in this setup. The influence of this Countermeasure would theoretically increase with larger Cache lines. Larger Cache lines can have both performance and security benefits, so we identify this as a desirable architecture change.

The ScatterRound Countermeasure was tested when implemented on a standalone context executing in 4-way mode, and as part of a dedicated setup with 4 contexts encrypting in parallel. Aside from fully preventing collision attacks in a round that uses 4 tables, we saw 225x more samples in isolated setup increasing 500x in shared setup, a factor 2,5x over just the shared processor. Both the Evict+Time and Prime+Probe attacks are influenced in this setup, since they have a similar effect to CacheSwap. Evict+Time required 160x as many samples in an isolated setup, and Prime+Probe 175x as 3/4th of its samples is invalidated. Overhead cost when in the 4-way configuration is 63.2%, which is more than Cache bypass of two rounds and thus infeasible. However, in a dedicated setup of 4 parallel encryptions the cost was 9.5%, compared to the AES throughput in that mode. However, the full overhead was 99% when compared to using the full processor in 8-way mode, twice as slow. This mode is especially interesting when using the ρ-VEX as a coprocessor, and could have additional security benefits because 4 parallel encryptions are mixed over Caches.

6

Conclusion

This section concludes the thesis on the Analysis of Cache Attacks and Countermeasure on the ρ-VEX Processor. In Section 6.1 a summary of the entire thesis is given. Section 6.2 answers the main problem statement and lists the main contributions. Section 6.3 list the future work that can be done on implementing Countermeasures against Cache Attacks on the ρ-VEX.

6.1. Summary

In Chapter 2, the necessary background to understand the rest of this thesis was given. The design of the Cache memory, and its location within the memory hierarchy of a computer is briefly introduced. The focus was to clarify the general concepts of Cache implementation in order to understand how these can be vulnerable to the Cache attacks in the later chapters. Then a overview of the general concept of the ρ-VEX processor was given, with focus on understanding how the run-time reconfiguration is implemented, and how the Cache of the system is designed to work with this reconfigurability. Finally the AES encryption algorithm is described. Giving a detailed description of the steps in AES-128 in order to understand how this algorithm can become vulnerable to Cache based attacks. Finally a fast implementation of the algorithm, the T-Table implementation, is discussed as it is the most common implementation on modern systems, and is the one studied in this thesis.

In Chapter 3, we first described how the Cache of a processor can pose a vulnerable side-channel. A broad overview of Cache attacks against AES in literature is given, alongside classifiers commonly used in literature. We identified a list of attacks that are implementable on the ρ-VEX in its current design. These attacks are the Evict+Time Attack, Prime+Probe Attack, Final Round Collision Attack, 3rd round Wide Collision Attack, Bernsteins Attack and First Round Trace based Attack. These attacks are described in detail to understand how they work and how a practical implementation would look like. We selected the Evict+Time, Prime+Probe and Final Round Collision attacks to test in this thesis. The other attacks were deemed to take too many samples to efficiently test multiple times, and trace based attacks were left out of the scope of this thesis. An overview of Countermeasures found in literature was given. These Countermeasures were divide into the categories Code modifications, Reducing System Level Privileges, Cache Redesigns, Noise based Countermeasures and Attack Detection. Then, the Runtime Reconfiguration system of the ρ-VEX was analyzed, identifying the possibility to influence execution time through lane sizes assigned to a process, and the possibility to influence access behaviour by moving contexts between different caches. We proposed five Countermeasure concepts: Causing timing noise by changing configuration sizes, generate access noise within a single shared ρ-VEX processor, efficiently generate access noise in higher Cache levels shared with other processors, prevent internal Cache collisions within a vulnerable algorithm and finally implement efficient systems that prevent Cache sharing between processes. We additionally identified the shared memory system between contexts to be a potential source of measurement noise.

In Chapter 4, implementations of the Cache Attacks Prime+Probe, Evict+Time and Final Round Collision on the ρ-VEX are described. Then, three Countermeasure concepts proposed in Chapter 3 are

worked out in three Countermeasure implementations called n-Lane, CacheSwap and ScatterRound. Practical details required to implement the Cache Attacks and Countermeasures are first introduced, describing the assumptions on reconfiguration privilliges, how to construct access attacks against the Cache architecture, how to determine reconfiguration words and how random reconfigurations as a Countermeasure are implemented in code. Our implementations of Evict+Time, Prime+Probe and Final Round Collision are described in detail. Then, three Countermeasure are introduced, describing their design, implementation, and potential effect on the attacks under certain circumstances. We first describe n-Lane: Noise via random configuration size variations. The amount of execution lanes is randomly changed during execution. The practical issue of losing cached data between configurations is discussed, and the potential for this Countermeasure to disturb the Cache state of the context is described. Finally the code implementation of the Countermeasure is described. We decide to test multiple configuration patterns possible within this Countermeasure, varying with what Caches are used and what reconfiguration percentages are used. Then, we describe *CacheSwap:* access noise via lanegroup swaps of two contexts. Another process is briefly swapped with the attacked process, so that the assumed Cache state is disturbed. The potential effects this Countermeasure can have on specific Cache Attacks is discussed, as it should influence all our attacks in unique ways. The importance of another context sharing the processor is highlighted. Finally we introduce the Countermeasure *ScatterRound:* preventing internal collisions via spreading operations over multiple Caches. This Countermeasure combines the potential of preventing collisions that are used for collision attacks and moving certain data lookups to a different Cache, by moving parts of the attacked rounds to isolated Caches on every encryption.

In Chapter 5, we did experiments on Cache Attacks and Countermeasures on a Genesys2 FPGA development board running the default ρ-vex design, with increased Cache capacity to benefit our attacks and Countermeasure performance. The standalone GRLIB version of the ρ-VEX system was ported to this board to use the on-board DDR3 memory chip. The OpenSSL AES implementation with Table T4 in the final round was ported to the system. We ran the attacker process and the AES code in a single process in a single hardware context, defaulting to a configuration that uses 4 of the 8 execution lanes. We made two setups. The first setup was a standalone setup where this process ran in isolation, and effects of a shared processor were partially emulated. In the second setup a second context executing a random selection of PowerStone benchmarks was running, with a configurable amount of stall cycles. This emulates a shared processor with varying workloads.

Executed in isolation, our attacks required comparable amounts of samples to that found in literature, but consistently less because of the smaller Cache lines and noiseless setup. Sharing the system with another workload significantly affected our timing attack efficiency, taking up to 200x as many samples for the Evict+Time and Final Round Collisions Attacks for the highest workload. The Access based Attack Prime+Probe also took up to 10x as many samples because of the shared memory system.

The n-Lane Countermeasure was mainly tested with a 10% chance to randomly reconfigure between the three configuration sizes at the start of an encryption. The overhead cost was 8.2% in the isolated setup, and increased roughly linear when increasing this percentage. The required amount of samples increased 800x for the Evict+Time attack and 750x for the Final Round Collision attack. Although not intended, the Prime+Probe attack also required 3x as many encryptions for all correct samples, but had no significant influence on the success of the attack. In a shared processor, the overhead cost for the parallel workload was 11.7%. However, running the n-Lane Countermeasure on this setup had no additional benefit when compared to the baseline. We saw that when trying to optimize the performance, the noise was significantly reduced. The cost of the unoptimized version was considered acceptable.

The CacheSwap Countermeasure was mainly tested with a 10% chance to randomly swap Caches with another process, during the computation of one column of a protected AES round. The overhead cost was 9.5% in the isolated setup , and increased to a maximum of 24% for 60% CacheSwap. The overhead cost increased to 11.5% in a shared processor setup. The required amount of traces increased with 40x for Evict+Time, 50x for Final Round Collision, but Prime+Probe was unaffected for lower percentages of CacheSwap, and only increasing with a factor 8x for the higher CacheSwap percentage 50%. When testing on the shared processor, the noise from the Countermeasure became for significant. 10% CacheSwap caused 8.94% extra execution cycles for the workload. For a high workload with 1k stall cycles, Evict+Time required 800x as many samples as the baseline, 4x as many

as the unprotected shared setup. A lower workload with 1m stall cycles the increase was comparable. The Final Round Collision required 225x with a medium workload. The 50% CacheSwap against the Prime+Probe attack made the trace requirement increase from 8x to 25x as many in this setup. The influence of this Countermeasure would theoretically increase with larger Cache lines. Larger Cache lines can have both performance and security benefits, so we identify this as a desirable architecture change.

The ScatterRound Countermeasure was tested when implemented on a standalone context executing in 4-way mode, and as part of a dedicated setup with 4 contexts encrypting in parallel. Aside from fully preventing collision attacks in a round that uses 4 tables, we saw 225x more samples in isolated setup increasing 500x in shared setup, a factor 2,5x over just the shared processor. Both the Evict+Time and Prime+Probe attacks are influenced in this setup, since they have a similar effect to CacheSwap. Evict+Time required 160x as many samples in an isolated setup, and Prime+Probe 175x as 3/4th of its samples is invalidated. Overhead cost when in the 4-way configuration is 63.2%, which is more than Cache bypass of two rounds and thus infeasible. However, in a dedicated setup of 4 parallel encryptions the cost was 9.5%, compared to the AES throughput in that mode. However, the full overhead was 99% when compared to using the full processor in 8-way mode, twice as slow. This mode is especially interesting when using the ρ-VEX as a co-processor, and could have additional security benefits because 4 parallel encryptions are mixed over Caches.

6.2. Main contributions

The main problem statement stated in the introduction of this thesis was:

Can the Runtime Reconfigurability of the ρ-VEX processor be used to implement efficient Countermeasures against Cache-based Side-Channel Attacks?

The Runtime Reconfigurability can be used to implement three types of Countermeasures in a standalone processor setup. The first is to use the variable amount of execution lanes assigned to a process to generate timing noise during execution. The second is to use the ability to swap executing contexts between Caches to cause extra Cache misses and random evictions. The final Countermeasure is to split memory lookups that can cause Cache collisions over multiple Caches in order to prevent them.

We selected three attacks to experiments Countermeasures against, the Evict+Time Attack, Prime+Probe Attack and Final Round Collision attack. We implemented timing noise through configuration size variations in a Countermeasure called n-Lane, implemented access noise through swapping contexts in a Countermeasure called CacheSwap and implemented preventing Cache collisions in a Countermeasure called ScatterRound.

The Genesys2 FPGA development board was used to instance the ρ-VEX processor. The OpenSSL AES implementation was ported to the ρ-VEX, and ran in two setups. The first as an isolated process, and the second in a shared processor where a workload consisting of randomly selected PowerStone benchmarks executed in parallel. All setups run in a default 4-way setting, using half of the processor resources to do AES encryptions.

10% n-Lane increased the required samples for a successful attack 800x for the Evict+Time attack, 750x for the Final Round Collision attack and a negligible amount for the Prime+Probe attack as a side effect. This Countermeasure added an average 11.7% of extra execution cycles per encryption, and added an average of 8.2% extra execution cycles to a workload sharing the processor, parallel to the encryption context. Running with this parallel workload did not make the attack require more samples.

10% CacheSwap running in a setup that forced Cache misses on every swap by clearing the other Caches of the system during runtime increased the required samples for a successful attack 40x for the Evict+Time attack and 50x for the Final Round Collision Attack. This Countermeasure added an average 11.5% of extra execution cycles per encryption, and added an average of 8.94% extra execution cycles to a workload sharing the processor. Sharing the processor with a parallel workload increased the sample size increase because of 10% CacheSwap to 800x for Evict+Time and 225x for Final Round Collision. Only higher percentages of CacheSwap had a notable effect on the Prime+Probe attack. The sample size requirement increased 8x for the isolate setup, and 25x in the shared processor setup with the 50% CacheSwap Countermeasure. The Countermeasure was considered to be a potential good Countermeasure for access based attacks if the Cache Line Size of the processor would be increased.

The ScatterRound Countermeasure prevented Cache collisions, making collision attacks impossible or more difficult based on how many collisions were prevented. Countermeasure had the additional benefit of generating noise similar to the CacheSwap Countermeasure, increasing the required samples for a successful attack 160x for the Evict+Time attack, 175x for the Prime+Probe attack that was not adapted to scan multiple Cache addresses and 225x for the Final Round Collision Attack, increased to 500x for the shared processor setup. We have seen that the performance costs were too great, 63.2%, when implemented in a regular, single encryption process on a shared processor. We thus suggested to run this Countermeasure on a setup where the ρ-VEX processed four encryptions in parallel, and saw that within this setup, the performance cost was 9.5%.

The main contributions of this thesis can be summarized as:

A general analysis of the vulnerability of the ρ-VEX has been done
We have shown that the current reconfigurable Cache design used on the ρ-VEX does not pose a specific threat to Cache attacks that require knowledge on the mappings of the data to the Cache, as these attacks can easily be adapted to work in any configuration size. We have seen that the Cache line size of just one 32-bit word in the data Cache made the attacks very efficient. We were able to retrieve a full key with just a one round attack in the Prime+Probe and Evict+Time attacks, and measured very low sample size requirements for successful attacks when compared to that reported in literature. We fully analyzed the security risk if an attacker can reconfigure at will. They can briefly reconfigure to execute select parts of an attacked algorithm in a isolates Cache, making for very efficient attacks. A reconfiguration also allows access to the Cache of processes sharing a ρ-VEX processor, allowing for cross-context attacks within the same processor. Finally, an attacker can stall processes sharing the processor in order to minimize noise in timing measurements, or in order to target the start of an algorithm and freeze it after the first couple of operations. Finally, the shared memory resources between the hardware contexts was identified to potentially be a significant noise source when multiple contexts are running in parallel on the same ρ-VEX processor. It was then experimentally shown that a timing based attack could take up to 200x more samples to be successful in a shared processor.

A collection of Countermeasure ideas on the ρ-VEX architecture has been proposed, three of which have been implemented and tested
In this thesis, five directions to implement Countermeasures were proposed. Causing timing noise by changing configuration sizes, generate access noise within a single shared ρ-VEX processor, efficiently generate access noise in higher Cache levels shared with other processors, prevent internal Cache collisions within a vulnerable algorithm and finally implement efficient systems that prevent Cache sharing between processes. We have shown an implementation of timing noise through configuration size variations called n-Lane increased the amount of samples required for the timing attacks Evict+Time and Final Round Collision to around 800x more traces. An implementation of access noise through swapping contexts called CacheSwap achieved 800x more traces for Evict+Time and 225x more traces for Final Round Collision when executed in a shared processor. The effect on Prime+Probe was only strong for higher chances to swap, but the overhead for these percentages were considered too high. An implementation of isolating lookups that can be used in private Caches, called ScatterRound, had additional benefits aside from preventing collisions. It made our Evict+Time Attack take 160x, Prime+Probe Attack 175x and Final Round Collision Attack 225x as many samples to be successful. We have shown that the overhead associated with 10% n-Lane and 10% CacheSwap was reasonable, but ScatterRound was concluded to require a specific execution setup to achieve performance costs that are acceptable.

6.3. Future work

Implement larger Cache Lines in the ρ-VEX Data Cache and compare Countermeasure performance
Modern CPU's that implement their Cache Lines with multiple processor words are more resilient to Cache Attacks. It reduces the amount of information leaked on accessed memory addresses by the amount of bits used to index the line because an attacker can only conclude what line is accessed and not what specific address on this line. An additional benefit is that there is a higher probability a line is

accessed for a lookup that is not the targeted lookup, and thus more samples are required to get clear results in Cache Attacks.
By itself increasing the line size should complicate the attacks, but it should also be beneficial to the effect of Countermeasures that generate access based noise, especially the CacheSwap Countermeasure. The random evictions caused by the CacheSwap Countermeasure can be more intrusive since entire Cache lines are evicted. As the Cache Miss Penalty increases with larger Cache Line sizes, the timing noise generated by this Countermeasure also increases. The increased cost of Cache Misses also influence the overhead costs related to all the Countermeasures. Rating the Countermeasures in terms of performance and influence on the attacks should be done again if the Cache Line size is increased.

OS formalization that allows trusted processes to issue their own configurations
At time of writing, there is no formalization of the rules of who can reconfigure the system. With this in mind, we would recommend the Operating System to be designed in such a way to improve security and Countermeasure implementation potential. The first suggestion is to make sure only a select set of processes can reconfigure the system. This should contain at least a system in the OS that manages the configurations to maximize performance and trusted operations that request reconfigurations to protect their data. If there is no way an attacker can manipulate the configuration of the ρ-VEX, then the full scale of options we discussed in this thesis becomes accessible for the protected process. The second suggestion is to implement the configuration selection of the OS in such a way to facilitate the sudden reconfigurations done by the protected processes, in such a way that neither reconfigurations clash with each other. Think of a Countermeasure related brief reconfiguration preventing the system from switching to a desired configuration that maximizes performance for the current workload.

Test hardware adjustments to enable efficient implementations of the proposed Countermeasures
In this thesis, we made assumptions that made determining the configuration words for our Countermeasure reconfigurations trivial. However, in practice, there might be a need for a dynamic system that determines what reconfiguration word needs to be produced. One way in which we think this could be possible is to have hardware support to track the possible configurations and provide a candidate random configuration word based on the main reconfiguration, the requested type of random reconfiguration and a hardware based random number generator.
Another adjustment that could lead to more efficient implementations of these Countermeasures is to decouple the instruction lane mappings and the instruction and data Cache mappings individually. This would make it possible to briefly swap contexts between data Caches without performance penalties related to the instruction Cache. This can also lead to more flexibility in reassigning system resources and could lead to performance benefits if used smartly.

Test the ρ-VEX security and Countermeasure potential in more practical setups
In this thesis we emulated two practical setups to test our attacks on a standalone processor. However, in a practical setup a lot of other factors can play a role in rating these Countermeasures. Is the ρ-VEX fully accessible to user processes, and are L1 access attacks thus a vulnerability? What is the required throughput of the setup, and what amounts of slowdown is thus acceptable? Is the system using extra Cache levels? This would decrease miss penalties associated with the Countermeasures, but introduce the risk of a cross core attack. What other workloads share this processor, and are the overhead costs to those processes associated with the Countermeasures acceptable?

Bibliography

[1] A. Goldstein, *[ONLINE]*, Available: https://unsplash.com/photos/EUsVwEOsblE.

[2] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[3] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, “Side channel cryptanalysis of product ciphers,” in *Computer Security — ESORICS 98*, J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 97–110.

[4] D. Page, “Theoretical use of cache memory as a cryptanalytic side-channel.” *IACR Cryptol. ePrint Arch.*, vol. 2002, no. 169, pp. 1–23, 2002.

[5] ——, “Defending against cache-based side-channel attacks,” *Information Security Technical Report*, vol. 8, no. 1, pp. 30–44, 2003.

[6] O. Acıiçmez, W. Schindler, and Ç. K. Koç, “Cache based remote timing attack on the aes,” in *Cryptographers’ track at the RSA conference*. Springer, 2007, pp. 271–286.

[7] D. J. Bernstein, “Cache-timing attacks on aes,” 2005.

[8] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, “Differential cache-collision timing attacks on aes with applications to embedded cpus,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2010, pp. 235–251.

[9] C. Percival, “Cache missing for fun and profit,” 2005.

[10] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[11] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *24th USENIX Security Symposium USENIX Security 15)*, 2015, pp. 897–912.

[12] Y. Yarom and K. Falkner, “Flush+ reload: A high resolution, low noise, l3 cache side-channel attack,” in *23rd Security Symposium Security 14)*, 2014, pp. 719–732.

[13] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+ flush: a fast and stealthy cache attack,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[14] Z. Wu, Z. Xu, and H. Wang, “Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud,” *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 603–615, 2015.

[15] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-vm side channels and their use to extract private keys,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 305–316.

[16] B. Gulmezoglu, M. S. İnci, G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross-vm cache attacks on aes,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 211–222, 2016.

[17] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 494–505. [Online]. Available: https://doi.org/10.1145/1250662.1250723

[18] ——, “A novel cache architecture with enhanced performance and security,” in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 83–93.

[19] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, “Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters,” in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[20] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding, “Scadet: a side-channel attack detection tool for tracking prime+ probe,” in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[21] C. Rebeiro, D. Selvakumar, and A. Devi, “Bitslice implementation of aes,” in *International Conference on Cryptology and Network Security*. Springer, 2006, pp. 203–212.

[22] R. Könighofer, “A fast and cache-timing resistant implementation of the aes,” in *Topics in Cryptology – CT-RSA 2008*, T. Malkin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 187–202.

[23] S. Gueron, “Intel advanced encryption standard (aes) new instructions set,” 2010.

[24] S. Wong, T. van As, and G. Brown, “ρ-vex: A reconfigurable and extensible softcore vliw processor,” in *2008 International Conference on Field-Programmable Technology*, 2008, pp. 369–372.

[25] S. Wong and F. Anjam, “The delft reconfigurable vliw processor,” *system*, vol. 1, p. 3, 2009.

[26] Hewlett-Packard Development Company, L.P., *VEX Toolchain*, https://www.hpl.hp.com/downloads/vex/, 2009.

[27] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[28] A. Brandon and S. Wong, “Support for dynamic issue width in vliw processors using generic binaries,” in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 827–832.

[29] J. van Straten, “A dynamically reconfigurable vliw processor and cache design with precise trap and debug support,” Master’s thesis, Delft University of Technology, Delft, Netherlands, May 2016.

[30] M. Yousaf, “Exploiting the reconfigurability of ρ-vex processor for real-time robotic applications,” Master’s thesis, Delft University of Technology, Delft, Netherlands, August 2016.

[31] J. Johansen, “Implementing virtual address hardware support on the ρ-vex platform,” Master’s thesis, Delft University of Technology, Delft, Netherlands, February 2016.

[32] L. van Bremen, “ρ-vex on chip: The design of an asic for a dynamically reconfigurable vliw processor with 24-port register file,” Master’s thesis, Delft University of Technology, Delft, Netherlands, August 2017.

[33] J. Daemen and V. Rijmen, “Aes proposal: Rijndael,” 1999.

[34] M. Neve and J.-P. Seifert, “Advances on access-driven cache attacks on aes,” in *Selected Areas in Cryptography*, E. Biham and A. M. Youssef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 147–162.

[35] O. Aciçmez and Ç. K. Koç, “Trace-driven cache attacks on aes,” 2006.

[36] J. Bonneau and I. Mironov, “Cache-collision timing attacks against aes,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, pp. 201–215.

[37] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx,” in *26th USENIX Security Symposium USENIX Security 17)*, 2017, pp. 51–67.

[38] O. Aciiçmez, “Yet another microarchitectural attack: exploiting i-cache,” in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[39] J.-F. Gallais, I. Kizhvatov, and M. Tunstall, “Improved trace-driven cache-collision attacks against embedded aes implementations,” in *International Workshop on Information Security Applications*. Springer, 2010, pp. 243–257.

[40] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, “Breakthrough aes performance with intel aes new instructions,” *White paper, June*, p. 11, 2010.

[41] J. Blömer and V. Krummel, “Analysis of countermeasures against access driven cache attacks on aes,” in *International Workshop on Selected Areas in Cryptography*. Springer, 2007, pp. 96–109.

[42] D. Page, “Theoretical use of cache memory as a cryptanalytic side-channel.” *IACR Cryptol. ePrint Arch.*, vol. 2002, no. 169, pp. 1–23, 2002.

[43] Y. L. Amd, L.-T. Lo, G. R. Watson, and R. G. Minnich, “Car: Using cache as ram in linuxbios.”

[44] R. Martin, J. Demme, and S. Sethumadhavan, “Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12. USA: IEEE Computer Society, 2012, p. 118–129.

[45] R. E. Kessler and M. D. Hill, “Page placement algorithms for large real-indexed caches,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[46] J. Shi, X. Song, H. Chen, and B. Zang, “Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 194–199.

[47] M. A. Mukhtar, M. Mushtaq, M. K. Bhatti, V. Lapotre, and G. Gogniat, “Flush+ prefetch: A countermeasure against access-driven cache-based side-channel attacks,” *Journal of Systems Architecture*, vol. 104, p. 101698, 2020.

[48] C. Reinbrecht, S. Hamdioui, M. Taouil, B. Niazmand, T. Ghasempouri, J. Raik, and J. Sepúlveda, “Lid-cat: A lightweight detector for cache attacks,” in *2020 IEEE European Test Symposium (ETS)*. IEEE, 2020, pp. 1–6.

[49] C. Li, C. Ding, and K. Shen, “Quantifying the cost of context switch,” in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, pp. 2–es.

[50] H. Aly and M. ElGayyar, “Attacking aes using bernstein’s attack on modern processors,” in *Progress in Cryptology – AFRICACRYPT 2013*, A. Youssef, A. Nitaj, and A. E. Hassanien, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 127–139.

[51] The OpenSSL Project Authors, *OpenSSL Cryptography and SSL/TLS Toolkit*, https://www.openssl.org/.

[52] Digilent, *Genesys 2 FPGA Board Reference Manual*, Available: https://reference.digilentinc.com/programmable-logic/genesys-2/reference-manual.

[53] Xilinx, *7 Series FPGAs Data Sheet: Overview*, Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[54] Xilinx, *ML605 Hardware User Guide*, Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/boards_and_kits/ug534.pdf.

[55] C. G. AB, *GRLIB IP Library User’s Manual*, Available: https://www.gaisler.com/products/grlib/grlib.pdf.

[56] A. Malik, B. Moyer, and D. Cermak, “A low power unified cache architecture providing power and performance flexibility,” in *ISLPED’00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, 2000, pp. 241–243.