UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Analysis of power leakages in transient execution on ARM architectures

Analyse des Stromverbrauchs in transienter Ausführung auf ARM Architekturen

Master Thesis

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Malte Stoffers

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 02. Januar 2020

Abstract

This master thesis analyses microarchitectural attacks on ARM-based platforms. Due to the latest developments in the field of IT-security, it is necessary to deal further with the subject of transient execution attacks. In concrete terms, what we want to investigate is the applicability of a power-based side channel in the context of a Spectre-type attack on the ARM architecture. We implement two variants of Spectre attacks and analyze the scope of the speculative context on the Cortex-A9 CPU. For power measurements we us the on board integrated FPGA as a software based approach. We test two sensor setups and analyze the different measurement results. The investigations show that, with an appropriate setting, especially ARMs conditional execution is possible to detect through a power trace. Still, the power measurement of transient executed code needs some further optimization to increase the detection rate and be useful in a real attack scenario.

Diese Masterarbeit beschäftigt sich mit mikroarchitektonischen Angriffen auf ARM-basierte Plattformen. Aufgrund der neuesten Entwicklungen im Bereich der Informationssicherheit ist es notwendig, sich weiter mit Angriffen auf die transiente Ausführung zu beschäftigen. Konkret untersuchen wir die Anwendbarkeit eines stromverbrauchsbasierten Seitenkanals im Zusammenhang mit einem Spectre-Angriff auf die Architektur des Herstellers ARM. Wir haben zwei Varianten von Spectre-Angriffen implementiert und den Umfang des spekulativen Kontextes auf der Cortex-A9 CPU analysiert. Für die Leistungsmessung wird der integrierte FPGA als softwarebasierter Ansatz verwendet. Wir testen zwei Sensoreinstellungen und erzielten unterschiedliche Messergebnisse. Die Untersuchungen zeigen, dass insbesondere die bedingte Ausführung durch den Stromverbrauch mit der zweiten Sensoreinstellung erkannt werden kann. Dennoch muss die Leistungsmessung von transient ausgeführtem Code noch weiter optimiert werden, um die Detektionsrate zu erhöhen und in einem realen Angriffsszenario nützlich zu sein.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 02. Januar 2020

Acknowledgements

First of all I would like to thank my thesis supervisor Prof. Dr. Thomas Eisenbarth from the Institute for IT Security at the University of Lübeck. The door to Prof. Eisenbarth was always open to me whenever I encountered a problem or had a question about my research. He always allowed this work to be my own, but always steered me in the right direction when he thought I needed it.
I would also like to thank Ida Bruhns for her help with the implementation of the Spectre attacks and Jonas Krautter for his support for installing the FPGA sensors.
Finally, I have to thank Claudius Pott, who always had helpful and goal-oriented ideas and patiently supported me over the last six months.

Contents

1 Introduction

Digitization is taking place in almost every application sector. To keep up with this rapid development and to guarantee the security of personal data, a wide range of research in the field of information security is required. Recent research showed a lot of new techniques to leak sensitive data using architectural-based vulnerabilities. To gain secret information attackers do not break cryptographic algorithms or search for implementation faults in software or protocols. Instead, attackers take advantage of the system architecture and behavior. This includes storage and memory isolation, caches and even the *out-of-order* execution of CPU instructions[KHF+19, LSG+18]. Microarchitectural attacks exploit functionalities of processor implementations and can compromise the security of computational environments even in the presence of sophisticated protection mechanisms like virtualization and sandboxing. Microarchitectural attacks can be considered as a special form of side-channel analysis. Side-channels aim to leak sensitive data like encryption keys. Examples are timing behavior, power consumption, and electromagnetic emissions. Different chip manufacturers such as Intel, AMD and ARM have different microarchitectures. ARM-boards are commonly used in embedded systems and smartphones and therefore are a popular target for attacks. A lot of research has already been attended on the ARM architecture. But some topics are still unexplored. This master thesis will investigate the so-called *Spectre attacks* in combination with a power side-channel on a platform based on ARMs architecture.

1.1 State of Research

Attacks on hardware can be sub-classified in side-channel and fault attacks. The main difference between them is that fault attacks are causing damage to the target device. This damage is either permanent (called invasive) or transient and disappears when restarted (semi-invasive). The attacker manipulates the environment of the victim's device which influences (targeted) computations. The aim is to either disable cryptographic security mechanisms or explore its secrets such as an encryption key. In [DLV03, BS97] faults are induced to recover secret keys of commonly used cryptosystems like RSA or AES.

In contrast, an attacker does not have to manipulate the environment but observe it for useful leakages. The first side-channel attacks have been discovered in 1996. In [Koc96] a

novel timing attack on RSA was proposed. This attack exploits implementation based timing behavior and could leak secret information. Since then, side-channel analysis became a huge research area in cryptography and a lot of attacks were discovered. Some examples are timing attacks, cache (timing) attacks like *flush and reload* [YF14], simple power analysis on *AES* key expansion [Man03] or differential power analysis [KJJ99].

Especially cache attacks were considered and are already well studied [LGS+16, YF14, OST06]. Because caches can contain secret data like encryption keys, they are are a promising target. Attacks on the processor cache analyze which addresses or data were used while executing a program by measuring access times of these addresses. If the time is low the address must be in cache, otherwise it is not and therefore the address was not used.

For different cache architectures and instructions some variants of cache attack algorithms exist. The most common and efficient are *Flush + Reload* [YF14], *Prime+Probe*, *Evict+Time* [OST06], *Evict+Reload* [GSM15] and *Flush+Flush* [GMWM16].

Power analysis was first described in 1999 by Kocher et. al [KJJ99]. Like timing attacks, it uses data and key-dependent fluctuations in observable power leakage. For example, secret-dependent control flow can have a different power consumption and therefore leak information of such a secret value. Usually, the power consumption is measured using an oscilloscope. Modern power analysis starts to use a *field programmable gate array* (FPGA) to measure the consumption indirectly [ZSZF13, GOKT16]. Some architectures include an FPGA on-chip with the CPU, and make it possible to remotely measure power consumption [SGMT18].

As a consequence of such attacks, research also started to focused on countermeasures for side-channel attacks. Microarchitectural side-channels leak changes to shared resources like caches caused by the execution of a victim's code. To mitigate this, one should eliminate data-dependent control flow over sensitive data [Koc96, Man03]. This results in data-independent timing behavior and power consumption. Some side-channels like acoustic or electromagnetic emissions are harder to mitigate since effective techniques require hardware isolation. Software-based countermeasures use obfuscation by generating additional and unpredictable noise [RLT15]. Countermeasures for more specific attacks have also been discovered. For example countermeasures against shared memory attacks of the TrustZone [AL17] or attacks based on hardware-software co-design [LLLGCN16]. Additionally, a study of *shuffling* against side-channel attacks was proposed in [VCMKS12].

Demme et al. define a metric for measuring the information leakage caused by side-

channels [DMWS12]. They observed that all side-channel attacks rely on recognizing leaked execution patterns. This metric was labeled *side-channel vulnerability factor* and measures information leakage by examining the correlation between a victim's execution and the attacker's observations.

New microarchitectural attacks use the *out-of-order execution* as a software-based side-channel. They form a new class of attacks, named *transient execution attacks* [CBS+19]. In 2018 researchers from Googles *Project Zero*, Cyberus Technology and TU Graz independently discovered a new form of microarchitectural attack, known as *Meltdown attack* [LSG+18]. It takes advantage of the *out-of-order execution* to break the memory isolation between user and kernel space. This isolation is a security feature of today's operating systems which ensures that user programs cannot access each other's memory space or kernel space. Using meltdown one can read arbitrary kernel memory data like sensitive data and passwords. Another novel attack called *Spectre* also takes advantage of the *out-of-order* execution [KHF+19]. Precisely, Spectre induces a victim to speculatively execute operations that would not occur during normal program flow. These operations leak confidential data via a side- or covert-channel to the adversary. Kocher et al. observed that the speculative execution mitigates security mechanisms including countermeasures against cache timing and side-channel attacks.

Since many widely used CPUs perform *out-of-order execution* in order to overcome latencies of busy execution units, Meltdown and Spectre affect many devices, from smartphones to high-end server systems. Both attacks caused a deep impact on the current research area and a lot of variants are currently being discovered. [BSN+19] and [ABuH+] introduce a speculative code-reuse attack that uses port-contention as a side-channel to leak information on a simultaneously multi-threaded processor. Port contention happens when instructions are scheduled to execute on the same execution port (which represents an execution unit). The authors describe how this contention can be measured and used for spying arbitrary data from userspace [ABuH+]. In [SSLG18] Schwarz et al. demonstrate a remote Spectre attack with the AVX[1]-based covert-channel. This marks a paradigm shift from local to remote Spectre attacks. This means Spectre attacks must also be considered on devices that do not run any attacker-controlled code. [WMW19] uses out-of-order execution to hide malware. Since speculative execution is nearly impossible to observe, even for debuggers, malicious code can be hidden in the "speculative world". This makes static and dynamic malware analysis harder because the functionality is contained in seemingly unreachable code. Wampler et al. demonstrate this as a new

[1] Advanced Vector Extension to the x86 instruction set architecture

kind of malware that evades existing reverse engineering and binary analysis techniques. Also the attack called *foreshadow* on Intels SGX relies on *out-of-order execution* [VBMW+18]. Foreshadow demonstrates how speculative execution can be exploited for reading the contents of SGX-protected memory as well as extracting the machine's private attestation key. Instead of accessing the SGX enclaves an attacker can control the surroundings.
Newer transient execution attacks focus on buffers which may leak data via transient execution. One is called *Zombieload* [SLM+19]. The authors uncover a novel meltdown-type effect in the processor's fill-buffer logic. They show that faulting load instructions may dereference unauthorized destinations previously brought into the fill buffer and report data leakage of recently loaded stale values across logical cores.
Microarchitectural attacks depend on the victims' hardware architecture. Intel CPUs are vulnerable against Meltdown, but most from AMD and ARM are not [LSG+18]. Spectre does not have such restrictions on the architecture, but especially some ARM-based CPUs do not support out-of-order execution or do not have a cache that can be used as a side-channel. ARM processor series[2] differ in such design decisions. Also, the instruction set makes it more challenging to mount e.g. cache timing attacks, as many do not have a *flush* instruction. Lipp et al. discusses this in [LGS+16]. They showed how cache attacks can also be applied on various ARM-chips which are commonly used in modern mobile devices. They deal with different cache structures (e.g. level inclusiveness) and instruction sets. Another technique, discovered in [GRLZ+17], makes it harder to launch cache attacks. Some processors use *autolock*, which prevents (as a side effect) cross-core evictions of cache lines from inclusive last-level caches. This is done by protecting a given line in an inclusive cache from eviction if any higher cache level holds a copy of the line. This technique was developed to handle performance penalties for maintaining inclusiveness in the cache hierarchy [Wil12].

1.2 Goals of this Master Thesis

As aforementioned this thesis will deal with microarchitectural attacks on ARM-based platforms, precisely we will investigate *Spectre* attacks using power as a side-channel. Parts of this research area have already been covered, but some are still unexplored. These include Spectre attacks with the usage of a power consumption side-channel and the practicability of Spectre variants on ARMs microarchitecture.
The task is to analyze the practicability of using a power consumption side-channel for Spectre attacks. This means, instead of observing cache behavior, like in the original attacks [KHF+19], this thesis will analyze whether power consumption of speculatively exe-

[2] Cortex A-series (high performance), R-series (real-time processing) and M-series (low power/cost)

cuted operations can leak useful information or not. Further, we will analyze what attacks based on speculative execution can be applied to ARM platforms. The power measurements will be taken using an FPGA. Thus, this thesis will use a Cortex-A platform with an integrated FPGA.

1.3 Approach

The structure of this thesis is as follows. In the next chapter, we will provide all necessary background information including a transient execution model, power analysis, and the description of the used Cortex-A9 platform. This CPU supports out-of-order execution and contains caches. The Digilent *Xilinx PYNQ-Z1* will be (mainly) used in this part. It is designed to be used with a new open-source framework that enables embedded programmers to utilize the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to design programmable logic circuits. In Chapter 3 we will implement the Spectre attack on the ARM architecture. The board uses the *ARMv7-A* instruction set and has no *flush* instruction. Therefore, cache lines have to be *evicted* using an *eviction strategy*. Fortunately, algorithms for eviction based cache attacks have been implemented and evaluated in [LGS+16]. Since *out-of-order* execution is an optional feature, we have to ensure it is activated. The speculative world is obviously bounded by some amount of time before the processor recognizes the faulty speculative execution. Therefore, the amount of time in the speculative world must be figured out. The next part (Chapter 4) will analyze the accuracy of the FPGA sensor and evaluate instructions and data-dependent computations to become part of a later attack possibility. With the Spectre attack working, we will perform observable instructions and data-dependent computations in the speculative world and measure the power consumption and investigate the applicability of power analysis combined with the Spectre attack.
In the last chapter, we will conclude the results and discuss possible improvements and future work topics.

2 Background

This chapter provides all necessary background information to follow the approach in the chapters afterward. First, we will describe basic architectural techniques to improve performance used in modern CPUs which cause a novel attack method. Then, we will explain how side-channel attacks and power analysis can be useful. At the end of this chapter, we will give an overview of the used ARM platform and describe briefly how FPGAs can be used for power analysis.

2.1 Transient Execution

Computer systems are described in detail by their architecture, precisely by the *instruction set architecture* (ISA). The ISA defines available registers, the addressing mode, the execution model, and the instruction set. The implementation of the ISA is called microarchitecture [CBS+19]. This includes e.g. pipeline depth, execution units, interconnections of system elements, the cache architecture, and branch prediction units. Both, the ISA and the microarchitecture are stateful, but only the ISA is visible to the user and programmer. The microarchitecture is hidden. The ISA's state is defined among other elements for instance by the data in registers. The state of the microarchitecture includes the content of the caches, the usage of execution units or the translation lookaside buffer.
Nowadays, it is difficult to increase the performance of computer systems. Due to physical limits, the speed and the number of transistors have not changed significantly in the last decade (quantum computation techniques excluded), instead CPU manufactures focused on adding more cores and optimize the pipeline by using parallelism and out-of-order execution. The intention is to keep the pipeline full at all times and avoid stalls, e.g. caused by a dependency between two consecutive instructions. Thus, the microarchitecture needs to be able to predict control and data dependencies. Microarchitectures implement prediction units and out-of-order designs for this purpose and can execute instructions in parallel. To avoid memory latencies most CPUs implement caches, which are small but fast storages.
Modern CPUs split instructions of a program into *micro operations* (μOPs) which are then being processed by the pipeline. The pipeline itself is split into different stages. Most manufacture implement at least a fetch, decode, and execution stage. To increase performance the pipeline is filled with μOPs which may be out-of-order. To save the correct order, the

CPU uses a so-called *re-order buffer* (ROB). The CPU has to re-order the μOPs correctly and decide whether they are committed to the architecture or not. Those out-of-order executed μOPs which are not committed are called *transient instructions*[LSG+18].
Despite out-of-order execution, many CPUs also implement prediction units for executing some instructions speculatively. This technique is used to avoid latencies of branches or data dependencies. A CPU predicts the outcome of a branch or data dependencies and continues executing instructions of the predicted path. Until the correctness of the prediction is verified the results are buffered in the ROB. If the prediction was right the result will be committed, otherwise, the CPU has to perform a roll-back.

2.2 Transient Execution Attacks

Transient execution attacks are a novel class of attacks. They exploit exception and branch misprediction events that may leak secret information in the microarchitectural state of the system. These attacks use the *out-of-order* execution as a software-based side-channel. While the architectural state, e.g. registers, is restored in case of a misprediction event, the microarchitectural state remains unchanged.
Transient execution attacks always follow the same procedure. 1) Initially, the attacker has to bring the microarchitecture into a suitable state, e.g. by cash eviction and manipulating branch prediction units. 2) In the second step, the attacker executes a trigger instruction which causes the victim's program to enter the transient execution of subsequent instructions until the trigger instruction is completed. 3) The third step includes the transient instruction sequence which acts like a microarchitectural covert channel by e.g. loading data into the cache. 4) Next, the CPU discovers the misprediction of the trigger and performs a roll-back of the architectural state (by flushing the pipeline, etc.). 5) In the end, the attacker can reconstruct the data via the covert-channel used in the transient execution phase [CBS+19]. The procedure is displayed in Figure 2.1.

Meltdown and Spectre attacks describe the two types of transient execution attacks discovered so far. Both exploit different CPU properties. The former exploits data from a faulting instruction which is forwarded to instructions ahead in the pipeline. Spectre, on the other hand, is based on control or data flow prediction techniques [KHF+19, LSG+18].

2.2.1 Meltdown

Meltdown uses the out-of-order execution to break security policies, e.g. memory isolation between user and kernel space. Memory isolation is a security feature of today's operating systems which ensures that user programs cannot access each other's mem-

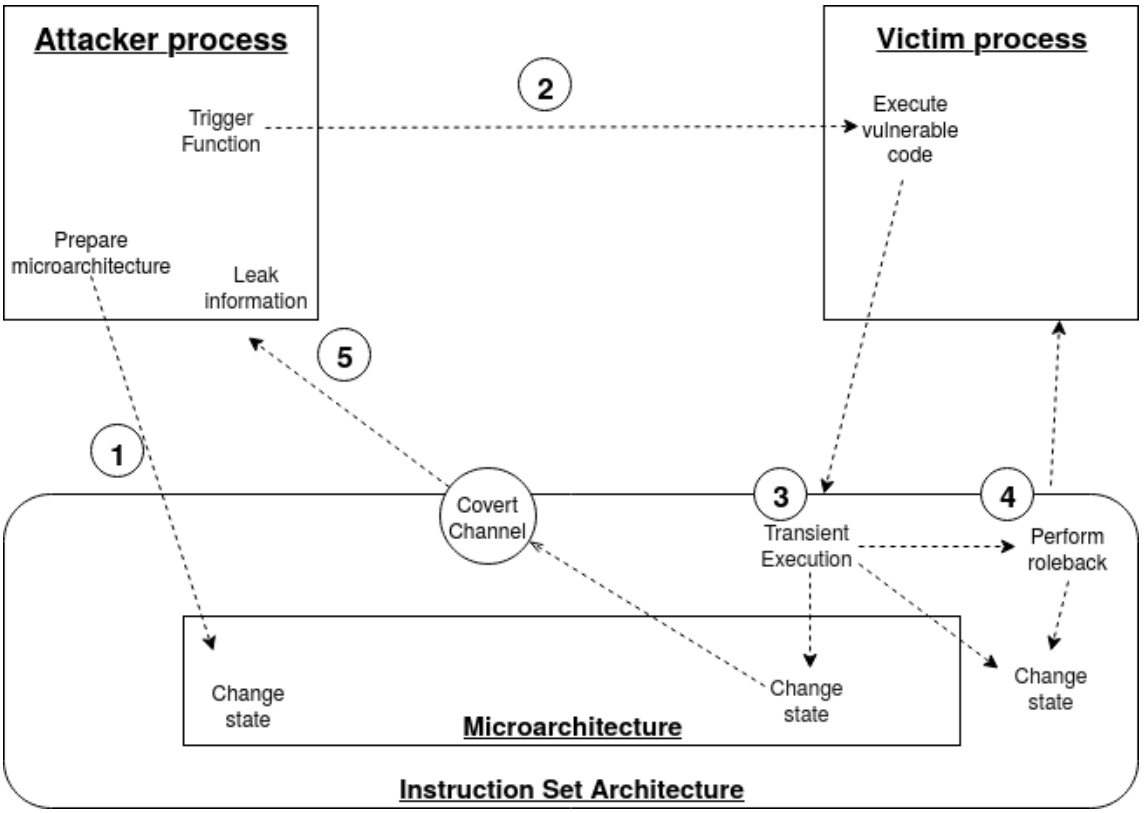


Figure 2.1: Transient execution attack procedure. 1) Prepare microarchitectural state, 2) Trigger victim, 3) Begin of transient execution which leaks data to the covert channel, 4) Perform roleback because of the misprediction, 5) Recover secret from covert channel

ory space or kernel space. Precisely, meltdown exploits transient execution following a faulting instruction and thus transiently bypasses *hardware-enforced* security policies to leak data that should always remain architecturally inaccessible for the application. Using this attack one can read arbitrary kernel memory data, e. g. passwords. For meltdown there exist some variants. The first attack reads kernel memory from userspace [LSG+18]. Foreshadow targets Intel SGX technology and uses a page table exception to bypass virtual translation [VBMW+18]. Other meltdown type attacks aim to read system privileged registers, floating-point registers or the fill buffer logic (attack named ZombieLoad) [SLM+19, CBS+19]. Since this work will only consider Spectre type attacks, we will not go into detail any further.

2.2.2 Spectre

The second type of transient execution attack was named *Spectre* and can, in contrary to meltdown, only compute on data that the application is allowed to access architecturally. Spectre bypasses *software based* security policies. Spectre uses to control and data flow predictions to access arbitrary memory addresses of the victim's system.
Considering only the microarchitectural root source that triggers a misprediction, there exist 4 different variants.

Spectre-PHT Kocher et. al [KHF+19] proposed the first two Spectre attack variants. Spectre-PHT exploits the *pattern history buffer* (PHT) which is used by the CPU to predict outcomes of a conditional branch. Consider the following code:
`if(x < size(array)){y = array[x]}`. If the condition is *false* but the PHT predicted the outcome to be *true*, the array fetch will be executed speculatively/transiently. At the moment the CPU realizes the misprediction, it discards the array fetch on the architectural state, but not on the microarchitectural. By observing the cache, for instance, we can reconstruct results of code snippets which would normally not be executed. However, due to the transient execution, we are able to read *out-of-bounds*. It is also possible to *write out-of-bounds*, as shown in [KW18].

Spectre-BTB The second variant is slightly different from the first one. Instead of using a conditional branch we now take advantage of indirect branches and therefore attack the *branch target buffer* (BTB). An indirect branch is taken speculatively if the target address of that branch is delayed. Meaning that the target address is not in cache or has not been computed yet. Contrary to the first variant we can now redirect the transient execution flow to an arbitrary address.

Spectre-RSB This variant exploits the *return stack buffer* (RSB). The RSB stores the most recent virtual addresses and pops the top for an executing `ret` (return) instruction.

This means, that the RSB predicts the return flow with the topmost address in the buffer. An attacker can manipulate the RSB by pushing an arbitrary address. When the victim transiently executes a *ret* the next address will be targeted from the attacker.

Spectre-STL Despite control flow prediction, some CPUs also can predict data flow. Store to load (STL) dependencies require that a memory load shall not be executed before write operation on the same address have completed. However, the CPU can predict which loads are independent of the write operation and execute them speculatively. If this is the case, we have again a transient execution that can leak information via the microarchitectural state. This *speculative store bypass* was proposed by Jann Horn [Hor18].

All Spectre attacks discovered so far use one of these root sources as a basis and then exploit different applications. These include for instance NetSpectre [SSL+19] where the authors propose an attack to read arbitrary memory over a network using transient execution behavior.
In addition to the root cause, the Spectre attacks can be sub-classified depending on the mistraining strategy. To train a specific branch we can consider different scenarios that depend on the *address space training* branch. First, the branch can be mistrained by the victim process or by an attacker process. Precisely, we can mistrain a branch using the *same-address-space* or a *cross-address-space*. Additionally, we can choose whether to use the vulnerable branch *itself* or use a *congruent* branch for achieving our objective. For more details, we refer to [CBS+19] in which all Spectre and meltdown variants were summarized and evaluated.

Since the discovery of transient execution attacks new attacks emerged fast. Portsmash, for instance, exploits the simultaneous multithreading. This attack takes advantage of port contention of execution units and creates a high-resolution side-channel [ABuH+]. This port contention happens if more processes want to use one execution unit which is accessible by a specific port. This master thesis will use the Spectre type attack and therefore no further description of more attack variants are necessary.

2.3 Side-Channel Attacks

Side-channel attacks were first introduced 1996 in [Koc96]. A side-channel leaks information caused by the implementation of an algorithm. It is not about weaknesses of the algorithm itself or the mathematical security, rather it is about weaknesses in the implementation. For example timing, power consumption or even sound can provide additional in-

formation for attackers. Timing or power behavior can leak whether instructions are being executed or not. For example, a common technique to implement modular exponentiation (necessary for RSA) is the *square and multiply* algorithm. This algorithm performs a square operation for each bit of the exponent if the bit is 1 the algorithm additionally executes a multiplication. Therefore using timing measurements one can reconstruct the exponent, which is in the case of RSA the private key [Koc96].
Covert-channels are a special use case of side-channel attacks. The attacker controls both the sender and the receiver and uses system components that are not intended for communications, e.g. writing and reading registers. They are often used for a proof of concept. In this master thesis we also use a covert-channel. The power leakage will be transfered from the FPGA sensor into *BRAM*. BRAM is a *block ram* which is a fast and small, internal memory that can be accessed each cycle[3].

Due simplicity, we will only use the term *side-channel* in all subsequent sections.

2.3.1 Cache Attacks

Cache attacks use timing behavior as a side-channel. An attacker measures the time of a memory fetch. If the access takes less time, then the data was already in the cache. This means the data has been used in earlier operations, otherwise, it would not be in the cache. This can be exploited to attack cryptographic algorithms. For example, Bernstein exploited the AES T-table implementation using cache attacks [Ber]. The difference of access timings of cached and uncached data is shown in Figure 2.2.
During the evolution of the cache architecture, different cache attack algorithms were explored [LGS+16, GSM15, YF14, YF14].

Evict+Time The algorithm steps are as follows

1. Measure the execution time of the victim program.
2. Evict a specific cache set.
3. Measure the execution time of the victim program again.

After the third step, an attacker can determine whether the cache set from step two was used or not. Since the cache set only contains the assigned data, the attacker can reconstruct which data was used while executing the victim's program.

Prime+Probe This method uses a different approach but gets the same results

1. Occupy specific cache sets.

[3] In contrary, DRAM is an external ram that is large, but has some overhead issues and also sends data back over multiple cycles.

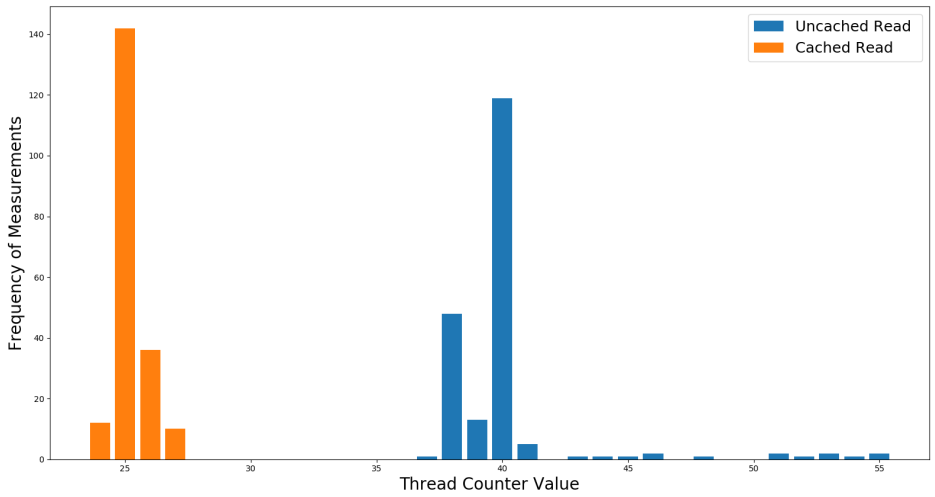


Figure 2.2: Memory access timing of cached and uncached data on the PYNQ-Z1 of 200 measurements. Time measurements are done with a thread counter and the CPU runs at 250MHz. X-axis: Measurement number, Y-axis: Timing as thread counter value

2. Victim program is scheduled.
3. Determine which cache sets are still occupied.

If the cache sets from step one are still occupied (and has a fast response time), then the victim processed data that was not loaded into this cache set. Otherwise, the victim used data that is assigned to the same cache set.

Flush+Reload *Flush+Reload* allows an attacker to determine which specific data is accessed by the victim.

1. Map a binary into address space.
2. Flush a cache line.
3. Schedule the victim program.
4. Check if the corresponding line from step 2 has been loaded by the victim program.

Flush+Reload requires shared memory. If the victim loaded the earlier flushed cache line, the attacker can determine which data was used.

Evict+Reload This attack is like *Flush+Reload*, but replaces the *flush* instruction by an eviction.

Flush+Flush The last method measures the execution time of the *flush* instructions. Gruss et al. [GMWM16] discovered that the execution time of a flush instruction depends on the fact whether data is in the cache or not. Thus, an attacker only needs to flush a cache line and determines whether the time needed was below or above a time threshold and can conclude which data was cached and which was not.

2.3.2 Power Analysis

The concept of power analysis was first described in 1999 by Kocher et. al [KJJ99]. With *simple power analysis,* (SPA) and *differential power analysis* (DPA), exist two main techniques to use power traces as an appropriate side-channel to leak processed data from the victim. Like timing attacks, it uses data and key-dependent fluctuations in an observable leakage, namely power. Nowadays, nearly all digital logic circuits are realized in CMOS technology (complementary metal-oxide-semiconductor). The power consumption of CMOS can be separated into a static and a dynamic part. The dynamic power consumption depends on the activity of different circuit parts, i.e. by changes in the control flow or processed data. And this dependency is the target of SPA and DPA attacks. SPA is used to reveal sequences of instructions by measuring the victim's code during the execution, for instance, a cryptographic operation. If the targeted code contains data or secret dependent control flow, an attacker might be able to reconstruct the secret using the power trace. To prevent SPA, developers can simply avoid conditional branches in which the outcome depends on secret information. DPA uses effects correlated to data values being processed. These fluctuations are mostly smaller and falsified by noise or measurement errors. However, DPA uses statistical mechanisms to extract useful leakage information of a power trace.
[BCO04] introduced a new form of power analysis. The authors showed how DPA can be combined with correlation to perform a more generic attack and uses template matching functions for calculating the similarity between two traces. This technique was named *correlation power analysis* (CPA).

Normalized Cross-Correlation

In this master thesis we will use the *normalized cross-correlation* for template matching. Cross-correlation is a measure of similarity between an image and a feature. In our case, the image is a power template and the feature will be a new and raw power trace. Cross-correlation is also known as a sliding inner-product and determines the time in which the two traces correlate best. For this purpose, one trace (feature or raw trace) will be gradually shifted over the second (image or template) [Lew01]. A *correlation coefficient* can be used to overcome some difficulties by *normalizing* the image and feature vector to get

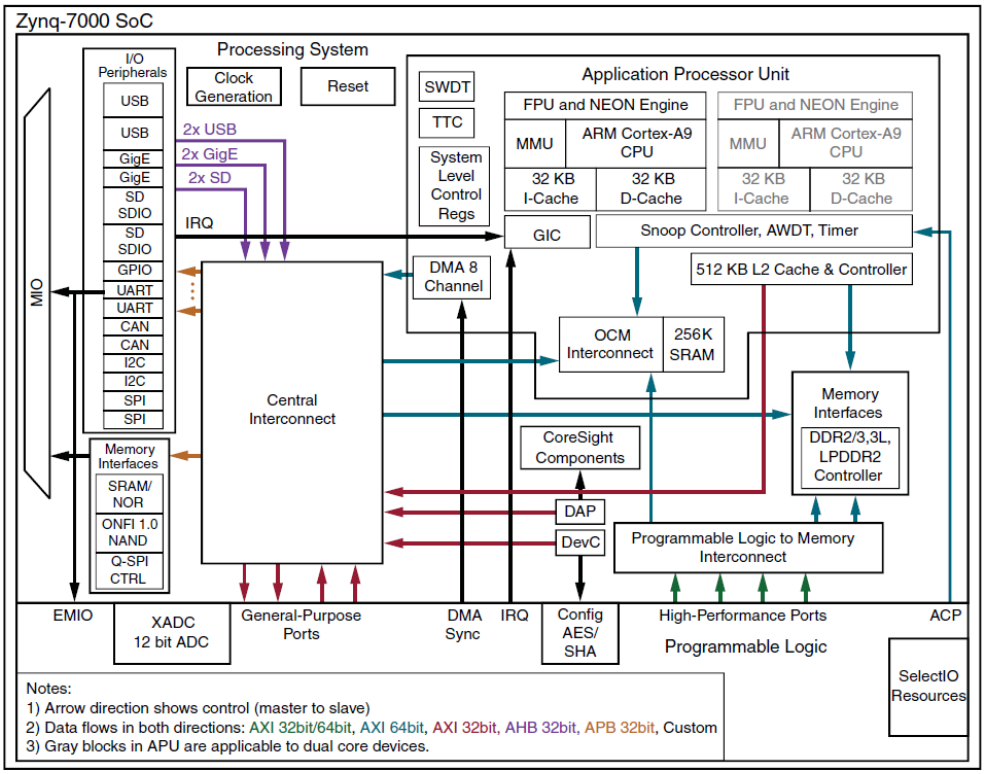


Figure 2.3: ZYNQ-7000 SoC Overview [XIL18]

a time-dependent correlation. We will use this technique as *normalized cross-correlation* as provided in the sensor framework.

Due to simplicity, we will not go into details. Nevertheless, for completeness we describe briefly the mathematical term of the cross-correlation

$$c(u,v) = \sum_{x,y} f(x,y)t(x-u,y-v)$$

as described in [Lew01]. The image is denoted by f, meaning our power template, and t holds the feature (meaning a new power trace) at position u, v. The correlation is the sum over all x, y of f. The maximum similarity of two traces has a correlation value of one.

2.4 PYNQ-Z1 / XILINX ZYNQ-7000SoC

The PYNQ-Z1 from XILINX is used for the following experiments. PYNQ-Z1 refers to the specific board and *ZYNQ* is the name of the architecture family from XILINX. Specifications of the board are shown in Table 2.1 and Figure 2.3 gives an overview of the board

architecture. The processing system (PS) and the programmable logic (PL) are on separate power domains, enabling the user of these devices to power down the PL for power management if required. The Zynq-7000 SoC is composed of the following major functional blocks [XIL18]:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnected
- Programmable Logic (PL)

Table 2.2 additionally describes the cache structure. Each of the two A9 processors have separate 32 KB level-1 instruction and data caches [XIL18, ARM16].

Table 2.1: PYNQ-Z1 Board specifications

CPU	32 Bit Cortex A9 dual Core
Architecture	ARMv7-A
Pipeline	out-of-order execution, 8-stages
Instructionsset	A32, NEON support
	VFPv3 floating-point engine (co-processor)
Caches	Seperate Level1 data and instruction cache, shared Level2 Cache

The size of a cache is the product of the line length, the number of sets, and the number of ways.

The PYNQ-Z1 implements static and dynamic branch prediction. Static branch prediction is provided by the instructions and is decided during compilation. Dynamic branch prediction uses the outcome of the previous executions of a specific branch to determine whether the branch should be taken or not. The dynamic branch prediction logic employs a global branch history buffer (GHB) which is a 4,096 entry table holding 2-bit prediction information for specific branches and is updated every time a branch gets executed. The functionality of a 2-bit predictor is shown in Figure 2.4. The edges describe whether the branch was taken or not. The nodes define the prediction itself. Using such a prediction mechanism we have to train a specific branch 2 times and can then be sure that this branch will predict *taken* in the next runs. Additionally, the board implements a branch target address cache (BTAC) which holds the target addresses of the recent branches. This

Table 2.2: Cache structure of the Cortex A9 processor [ARM16, XIL18]

	Level 1	**Level 2**
Size	32KB	512KB
Line length	32Byte	32Byte
Number of Sets	256	2048
Number of Ways	4	8
Replacement Policy	pseudo round-robin or pseudo-random	snoop coherency control utilizing MESI
Access	physically indexed and physically tagged	physically indexed and physically tagged
Additional information	supports two 32 byte line-fill buffers and one 32-byte eviction buffer	provides exclusive mode, implements a 16-entry deep preload engine for loading data into L2 cache memory, has multiple line-fill buffers

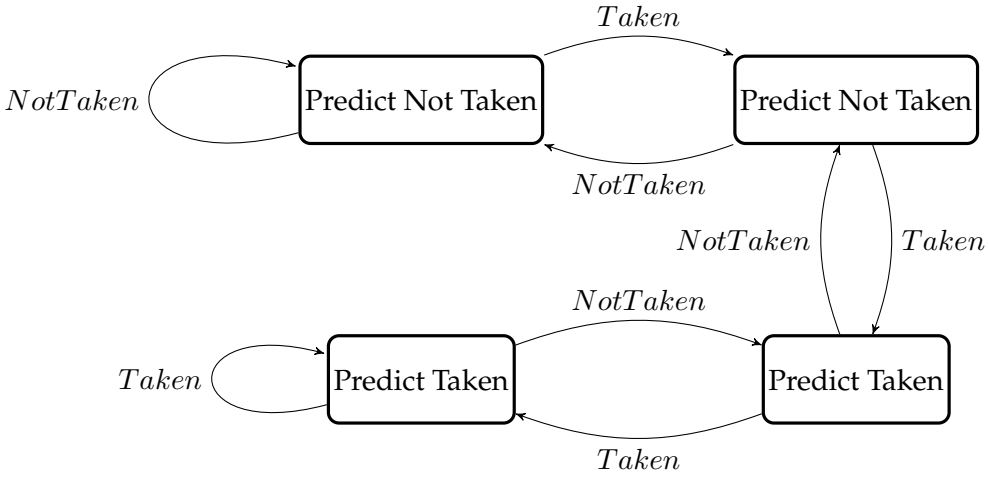


Figure 2.4: State Machine of a 2-Bit Branch Prediction Unit

512-entry address cache is organized as 2-way × 256 entries and provides the target address for a specific branch to the pre-fetch unit before the actual target address is generated based on the calculation of the effective address and its translation to the physical address. The Cortex-A9 CPU can predict conditional branches, unconditional branches, indirect branches, PC-destination data-processing operations, and branches that switch between ARM and Thumb state [ARM16, XIL18]. BTAC is used by instructions that store the target address (partially or completely) in registers e.g. the instruction *branch and exchange* (in assembly: `BX [r1]`). As also described in the reference manual the Cortex-A9 CPU employs speculative execution of instructions enabled by the dynamic renaming of physical registers into an available pool of virtual registers. The CPU employs this virtual register renaming to eliminate dependencies across registers without jeopardizing the correct execution of programs [XIL18].

2.4.1 Integrated FPGA

The PYNQ-Z1 has a *field programmable gate array* (FPGA) SoC. We will use this FPGA to measure power consumption. How this is done is described in the next section. This board uses the Artix®-7 FPGA logic from XILINX. The processing unit (CPU) and the FPGA can be tightly or loosely coupled using multiple interfaces and other signals that have a combined total of over 3,000 connections. Since this master thesis will not be focused on programming the FPGA, further details of the FPGA are not necessary, nevertheless, we give some key features in the following list. For more details, we refer to the reference manual [XIL18]. We use an existing logic circuit for power measurements described in the next section.

Key Features

- Configurable logic blocks (CLB)
  - 6-input look-up tables (LUTs)
  - Register and shift register functionality
  - cascadable adders
- 36 KB block RAM
  - Dual Port
  - Programmable FIFO logic
- Digital signal processing - DSP48E1 Slice
- Clock management

- Configurable I/Os
- Low power serial transceivers
- Integrated interface block for PCI Express designs

2.5 Power Analysis Using an FPGA

Usually, power analysis of integrated circuits is done using an oscilloscope connected directly to the target. Investigations of transient voltage fluctuations in FPGAs [GOKT16] offered a new method to power analysis. [SGMT18] showed that based on this voltage fluctuations it is possible to *remotely* use power as a side-channel.

The idea is to use a delay line, in which a clock signal propagates through a chain of buffers. Observations showed that the delay of these buffers depends on the supply voltage and is observable. At the moment when any other circuit on the same *power distributed network* (PDN), meaning the same power circuit, becomes active and consumes additional power, a voltage drop leads to slow down the buffers of the delay line. This results in a reduced numeric value in the *time-to-digital converters* register, which holds the number of toggled buffers. In [ZSZF13] it was shown that these TDCs can be used to sense variation in supply voltage based on the concept of measuring the propagation time of a signal.

Using this technique we can analyze power consumption indirectly over the implementation of the FPGA of the PYNQ-Z1. The implementation of the sensor on the FPGA was developed by Dennis Gnad and Jonas Krautter from Karlsruher Institute of Technology.
In this master thesis, we analyze two different sensor setups, each deployed and adopted by D. Gnad and J. Krautter. Both sensor setups were implemented in *VHDL* (very high speed integrated circuit hardware description language). The setups differ in the used frequency. To increase the accuracy of the sensor the speed of the processing system (CPU) and the programmable logic (FPGA) need to be as equal as possible. The first setup uses a CPU and FPGA frequency of $130\,\mathrm{MHz}$. Unfortunately, we discovered that using this setup a Spectre attack was not possible anymore. Therefore, J. Krautter reconfigured the setup, which uses a frequency of $250\,\mathrm{MHz}$ on the CPU and the FPGA. Both setups produced different results which are described in Chapter 4.

To use the FPGA sensor we use *Vivado 2018.3* to generate the *bitstream*, which is the compiled version of the sensor and load it to the FPGA. The sensor waits for a trigger signal on a GPIO pin and stores the values in *BRAM*. Since *BRAM* is accessible from user space,

we can load the values using a *C* program and can analyze them independently from the FPGA.

Additionally to the sensor, the developers delivered a program to read traces from the correct BRAM address and to analyze the generated traces using the following procedure.

1. Collect n traces of the targeted code sequences.
2. Calculate the average of the collected traces for all different code sequences. We will call the average traces *templates* in further parts of this thesis.
3. Collect n traces a second time, each randomly chosen from the targeted sequences.
4. For each new trace calculate the normalized cross-correlation to every template from step 2.
5. Choose the best correlation value and its linked code sequence and check the correctness of this detection.

3 Spectre Implementation

In this chapter we will implement the first two spectre variants, Spectre-PHT and Spectre-BTB.
Transient execution attacks including Spectre affect not only Intel and AMD processors but also those of ARM. ARM lists all processors and their vulnerability to Spectre variants on their website [Ltd]. This master thesis uses the PYNQ-Z1 board with a Cortex-A9 dual-core processor. We can implement two variants of Spectre, namely, bounds check bypass (Spectre-PHT) and branch target injection (Spectre-BTB), described in Section 2.2.2. The main difference between these two is that the first always executes the same code sequence. In the second we can jump to a target address of our choice and therefore have more options for executing code sequences. Nevertheless, both have in common that the attacker trains a specific branch in such a way that the victim is executing the destination address of that branch speculatively.

Libflush Library and Timing Setup

A simple way to force a program to execute code speculatively is to use the latency of memory access. Such a latency can be achieved by flushing specific cache lines. Since the *ARMv7-A* architecture provides no instruction to flush parts of the cache, we have to evict them indirectly.
For the cache eviction, we use the library `libflush`. It was implemented for cache attacks on ARM [LGS+16] and is available on GitHub[4]. This library calculates congruent addresses and stores them in so-called eviction sets. For flushing/eviction, the library accesses these memory addresses and overwrites data in the specific cache set. Using this library, one can also evaluate the best eviction strategy. The computation of an eviction set for one address takes approximately 10 seconds (measured on the PYNQ-Z1) with a frequency of $250\,\mathrm{MHz}$.

Despite cache eviction, we also have to be able to make timing measurements. Most architectures provide performance counters for their system. This includes registers holding the current cycle count and other event counters. However, these counters may not be accessible from user space, even with root privileges. The Cortex A9 CPU has some counters

[4] `https://github.com/IAIK/armageddon/tree/master/libflush`

```
1 // PMUSERENR = 1
2 asm volatile ("mcr p15, 0, %0, c9, c14, 0" :: "r"(1));
3 volatile unsigned pmusserenr;
4 asm volatile ("mrc p15, 0, %0, c9, c14, 0" :: "r"(pmusserenr));
5 printf("%u",pmusserenr);
6 // PMCR.E (bit 0) = 1, PMCR.C (bit 2) = 1
7 asm volatile ("mcr p15, 0, %0, c9, c12, 0" :: "r"(5));
8 // PMCNTENSET.C (bit 31) = 1
9 asm volatile ("mcr p15, 0, %0, c9, c12, 1" :: "r"(1 << 31));
```

Listing 3.1: Enabling Read-priviliges of the performance counters

```
1 volatile unsigned cnt_value;
2 asm volatile ("mrc p15, 0, %0, c9, c13, 0" : "=r" (cnt_value));
```

Listing 3.2: Read the cycle counter

too, among them a cycle counter. To use performance counters one can either use libraries that provide an interface or use them directly with appropriate assembly instructions. We will try both options. First, we use assembler instructions to use the cycle counter. On the PYNQ-Z1 the register `PMUSERENR` has to be modified to enable user access. This can be done using the inline assembly instructions in Listing 3.1. However, this is a control register that is only readable from userspace. Changing bits of a control register requires kernel privileges.

To read the cycle counter stored in register `PMCCNTR` one can use the code in Listing 3.2. As aforementioned the access to the counters has to be enabled by setting a bit called `PMUSERENR`. In the provided OS-image[5] the bit `PMUSERENR` is not set. Therefore we cannot use the performance counter without building our own image or kernel respectively. Instead of building our own kernel, we tried the performance library called `perf`[6]. But as expected this software also has no access to the counters.

By default, `libflush` uses performance counters with `perf` to do cache attacks which in general need timing behavior for observation. Using the `libflush` library we can use a *thread counter*. So instead of counting cycles, we create a thread that increments an integer as fast as possible. Due to OS scheduling and other influences, the thread counter may not be as reliable as a cycle counter. However, in [LGS+16] the authors evaluated the precision of such counters and came to the conclusion that a thread counter is a suitable measurement technique.

With a little trick, we can measure cycles approximately using an indirect approach. With

[5] `https://pynq.readthedocs.io/en/v2.2.1/getting_started/pynq_image.html`

[6] Perf manual on `http://man7.org/linux/man-pages/man1/perf.1.html`

```
1  //Access integer variable a, afterwards a should be cached
2  a = 42;
3  //Read performance counter
4  time = libflush_get_timing(session);
5  //Access the variable a
6  temp = a;
7  //Calculate the amount of time taken from this access
8  cached_time = libflush_get_timing(session)-time;
9
10 //Evict the cache line holding a
11 libflush_evict(session, &a);
12 //Determine the amount of time
13 time = libflush_get_timing(session);
14 temp = a;
15 uncached_time = libflush_get_timing(session)-time;
```

Listing 3.3: Determining cache timing behavior

this thread counter, we have a time unit that we can use to compare instructions, with respect to timing. Analyzing the Cortex-A9 technical reference manual we learned that the instruction `mov r0,r0` takes exactly one cycle. This instruction is comparable with a `NOP` on Intel architectures and is in the following described as a `NOP`[7]. To approximate the number of cycles taken by an instruction, e.g. a memory fetch, we measure the time of that instruction using the thread counter and determine how many NOPs need the same amount of time. The number of NOPs used is equal to the number of cycles taken by the instruction. Since the amount of time measured by the thread count is not always the same, we are using the average of 1000 measurements.

Cache Timings

For our attack, we need to know whether data is cached or not. This can be done by observing the timing of memory fetches. Therefore, we initially determine how much time access to cached and un-cached data takes. Our test code is displayed in Listing 3.3. First, we access the variable a to transfer it into the cache. Then we access a again and measure the time. To measure the access time of un-cached data we can just evict the variable and access it afterward.
The timings we have measured are shown in Table 3.1. The thread counter value is lower because the thread is not scheduled and executed every cycle. We can now calculate a threshold for cached and un-cached data on the PYNQ-Z1. Typically the threshold is

[7] ARM has also a NOP instruction, but this can be discarded by the pipeline. This means we can not be sure whether the instruction takes 1 cycle or no cycles.

Table 3.1: Cache Timing Measurements wit a CPU speed of 250 MHz and memory speed of 200 MHz

Instruction	Thread Counter (avg)	Cycles (approx)
Memory Fetch of un-cached data	41	35
Memory Fetch of cached data	26	10

calculated as follows:

$$(cachedTiming + uncachedTiming)/2$$

Therefore our cache threshold is $(41 + 26)/2 = 33.5$.
A graph of the timing measurements was already given in Figure 2.2. For the following implementations we will use the *evict+reload* algorithm (described in Section 2.3.1) for observing the cache contents.

3.1 Spectre-PHT

We have implemented a proof of concept code for *Spectre-PHT* based on the proposed example implementation for the x86 architecture in [KHF+19]. Our implementation differs in the following aspects.

- The used Cortex-A9 has no *flush* instruction, therefore we have to evict cache lines by accessing congruent addresses (see Section 2.3.1). As already mentioned we will use the *libflush* library for this purpose.
- To check whether code was speculatively executed or not we will use a timing covert-channel. For the actual attack, we will omit this part.
- For performance reasons, the loop counter and other variables can and will be decreased or deleted without violating the attack's logic and result.

The basic idea of this attack is to train the branch prediction unit to take a conditional branch speculatively and access a memory address that would not be fetched normally. Listing 3.4 shows an example branch which will be executed speculatively. Note that this code represents the logic of the spectre attacks, the array fetch is observable through the cache. Since we want to use power consumption as the leaking side-channel, this instruction might not be suitable for us. Instead of exploring data in memory and leaking this via the cache, our victim function should perform operations that result in different power consumption. We will analyze this topic in the next sections. The most difficult part of the attack is to train a specific conditional branch to behave exactly as we want.

```
1 if(x<arraySize){
2     //START of speculative context if x > arraySize
3     temp=array[x]
4     //END of speculative context
5 }
```

Listing 3.4: Bounds Check Bypass Code

In Section 2.2.2 we already described different scenarios to train a branch, considering the *address-space* and whether we train the branch *itself* or a branch at a *congruent address*. For simplicity and without loss of generality we will train a conditional branch *itself* in the *same address-space*. To train a branch we first have to check what kind of prediction units are used on the PYNQ-Z1 board. As described in Section 2.4 this development board uses a global history buffer (also named pattern history buffer) with 4096 entries holding 2-bit prediction information for used branches and is updated after each branch execution.
Therefore, we can train a conditional branch by executing the branch $n > 3$ times, meaning that the program flow will take the branch n times. At the $(n+1)'th$ execution the prediction unit will predict the branch to be taken, both when the condition is met and when it is not. To avoid a fast resolution of the condition we `evict` the variable `arraySize` and take advantage of the resulting memory latency.
Using this method we can force the CPU to execute code speculatively. Experimentally, we discovered that training a conditional branch four times is reliable to execute the fifth execution speculatively.
In Figure 3.1 we displayed the program flow of our implementation. This attack uses a loop amount of 30. In each round we calculate a variable `x` in such way that each fifth round `arraySize` is bigger than `arraySize`. In our implementation `arraySize` has a value of 16. To generate memory latency, we evict `arraySize` in each loop. Note, that the operation `temp=array[x]` is just an example that can be used for observation in the cache architecture.
To summarize, these are the steps of our implementation:

Step 1 Evict the whole array and set the loop counter to 29.

Step 2 Generate the value of x in such way that each fifth loop the variable holds e.g 170, otherwise a value between 1 and 15.

Step 3 Evict the variable $arraySize$.

Step 4 Execute the conditional branch.

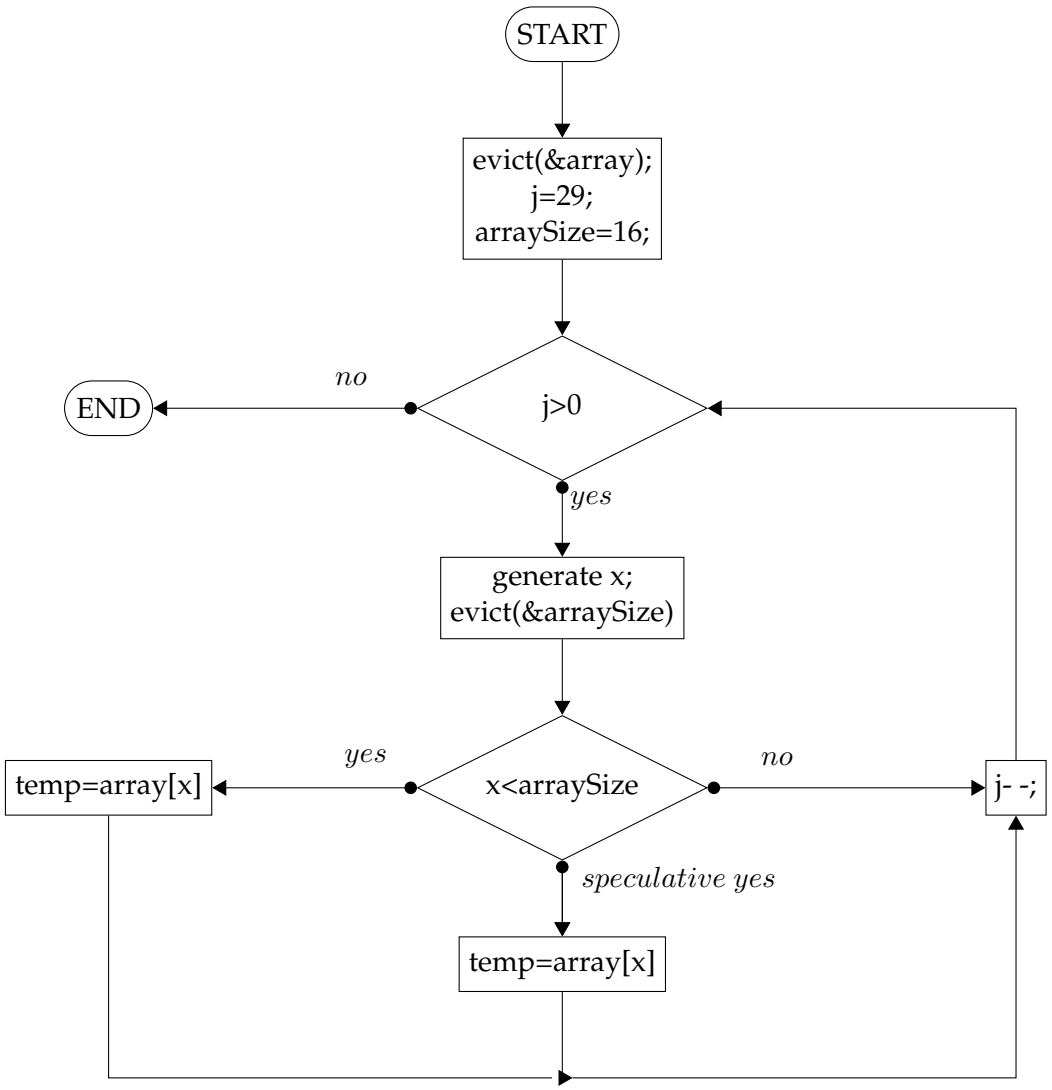


Figure 3.1: Programm Flow of Spectre-PHT Implementation. In every fifth loop the program will enter the speculative path.

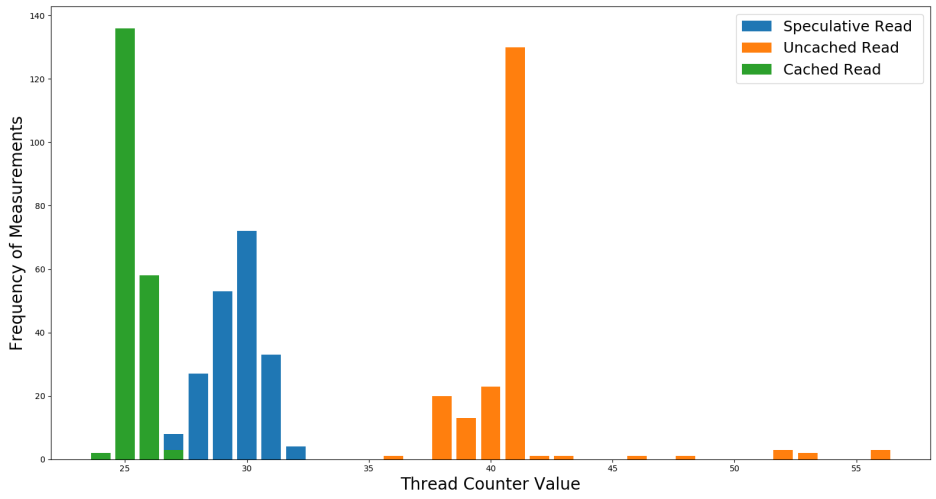


Figure 3.2: Timing of 200 measurements of an array fetch executed in transient/speculative execution. X-axis: Measurement number, Y-axis: Timing as thread counter value

In every fifth loop we will access the $array$ at index 170 which will be reset by the CPU because of the mis-prediction. However, we can observe this value was loaded speculatively by checking the cache timing of the used values (in our case `array[170]`).

The cache observation is shown in Figure 3.2. For checking the reliability of our attack implementation we executed it 100 times. As we can see, the transient array fetch, forced by the training described above, is cached. In the previous section, we calculated a threshold, which defines whether a memory address is already cached or not. We can recognize that the timing of the speculated fetch is continuously higher than a normal data fetch that is cached. The reason we can not be sure of, but we highly assume that the speculative fetch was loaded only in Level2 (L2) cache and not in the Level1 (L1) cache. We could test that by determining the speed difference between L1 and L2 cache. But since the measurements are always lower than our calculated threshold, we can be sure that the value is cached. Whether in L1 or in the L2 cache is not important to our purpose. We, therefore, refrain from further investigation of this behavior.

To summarize this section: Our results confirm ARM's information that *Spectre-PHT* is possible on ARM-Cortex A9 architecture [Ltd] and proofs the vulnerability on the PYNQ-Z1.

3.2 Spectre-BTB

The second variant of Spectre attacks manipulates the branch target buffer (BTB). This buffer is used to predict the addresses of *indirect branches*. As described in Section 2.4 the PYNQ-Z1 also uses a BTB and should be vulnerable to this attack. Indirect branches can, for example, be function pointers in C. For a proof of concept, we can manipulate a function pointer as an indirect branch, using the same approach of Spectre-PHT (see previous Section 3.1). We will train the branch four times to mispredict the outcome of the indirect branch in the fifth run. Again, we will use the *same-address-space* and the (indirect) branch *itself* for our mistraining strategy. The implementation steps are as follows:

Step 1 Set loop counter to 30

Step 2 Set the function pointer to *function1*, and only every fifth loop to *function2*.

Step 3 Evict the function pointer from the cache

Step 4 Jump to the destination of the function pointer

The main code part is listed in Listing 3.5. We first define an array containing function pointers of type `int`. Then we repeatedly set the function pointer `f_ptr` to the function defined in the array `table` and evict this pointer immediately afterwards to use memory latency. The functions `func1` and `func2` return each data of different memory addresses which we can observe via the cache. The *indirect branch* is executed in line 9 by calling the function pointed by `f_ptr`. Because of a misprediction, `func1` will be executed speculatively in every 5th loop. After detection of the misprediction, the architecture performs a rollback and executes `func2` (for the victim's view) correctly. Again, we executed this approach several times and used cache timing as a covert-channel to leak whether an array fetch was loaded falsely into the cache because of speculation. And just like in the previous implementation of the first variant we can conclude reliably that the specified array address has been cached speculatively. The timing measurements were similar to the one before which is why we are not going to add another graphic as proof and refer to Figure 3.2.

However, the intention of Spectre attacks is to execute code that would normally not be executed. This means we have to ensure that the execution differs in the speculative world to gain an advantage of the misprediction. To solve this we can use a function that expects an input, e.g. an integer. But for our experiments, Spectre-BTB has no advantage against Spectre-PHT. In both cases, we execute the same code and in speculation the also get the same values.

```
typedef int (*func_ptr)();
int (*f_ptr)();
const func_ptr table[] = {func1,func1,func1,func1,func2};
for(int i =0; i<30;i++){
    f_ptr = table[i%5];
    libflush_evict(lfs, &f_ptr);

    //Execute func1 speculatively in every fifth loop.
    int a = (*f_ptr)();
}
```

Listing 3.5: Spectre-BTB, exploiting the branch target buffer

The advantage of the second variant only becomes noticeable when we train the branch from another process to jump to an arbitrary address in the application. Thus, in our investigations for a proof of concept, we have no advantage over the use of the first variant. Therefore we will use the implementation of Spectre-PHT in the following sections.

4 Spectre With A Power-Based Side-Channel

In this chapter we will combine our spectre implementation and the power sensor (described in Chapter 2). Initially, we have to analyze the transient context to determine what can be done speculatively. Afterward, we will take power measurements to understand the range of possibilities using the FPGA sensor. In the later sections, we will combine our Spectre implementation and the FPGA sensor.

4.1 Speculative Context

The amount of time or instructions performed speculatively is of course limited. To find suitable code gadgets (vulnerable code sequences) we first have to determine what code can be executed in the speculative world. We force the program to execute code speculatively by evicting necessary data for a conditional branch. Therefore the time to load this evicted value and evaluate the condition is an upper bound. Additionally, the speculative execution is limited by the maximum number of speculative instructions (see [CBS+19]). Unfortunately, we could not find any information about this in ARMs reference manuals [XIL18, ARM14a, ARM16, ARM14b].

With the timing measurements in Table 3.1 we already have detected the first scope, a memory fetch takes approximately 35 cycles. We checked whether there is more time in the speculative context or not and what instructions can be executed. Since speculation is difficult to observe, calculating the extent of speculation is difficult too. A correctly executed speculation can be observed via the changed registers. Since the execution is correct, however, we would have to distinguish between correct speculative and normal execution. In other words, we would have to determine the point at which speculation passes into normal execution. However, we are interested in the wrongly predicted execution. This execution can no longer be seen in variables or registers because the processor's state is rolled back to the last valid state after detecting the wrong prediction.

A debugger is able to freeze the system and can display at least the architecture. This means we can observe for example register values. But, it is doubtful that a debugger can display the microarchitectural state, including the speculative execution step by step. Thus, we decided to skip the debugger part and use the additional time in later parts of

```
1 if(x<arraySize){
2 // Place here additional instructions, e.g. mov r0,r0
3         temp=array[x]
4 }
```

Listing 4.1: Testcode to investigate the speculative context

this master thesis.

So how does incorrect speculative execution manifest itself? As described in the original paper of Spectre attacks [KHF+19], we can see executions in the microarchitecture, especially in the cache. This means memory access is reset in the variables and registers but remains in the cache. To check if speculative execution is done we determine whether a specific value is in cache or not. Otherwise, we can not be sure if the code was executed speculatively.
In the following we want to determine which instructions can be executed in speculation. Since we can not test all available instructions of the instruction set, we will pick some representative ones which we might be able to use in a later attack possibility or are used for cryptographic operations. Therefore, we investigate the following instructions:

mov This instruction is used for writing *constants* or copy the content of one register into another one.

Arithmetic Operation We will test `add`, `sub` and `orr` to represent arithmetic operations and logical operations. These operations need an equal number of cycles and have the same input-output structure.

mul Multiplication differs from the above with respect to cycle amount. According to [ARM16], this instruction might have a delay in 1-4 cycles.

smull This is a *signed* multiplication that takes more output and input registers and has to be considered different from the normal multiplication. Remark that the instruction set has more different multiplication variants, but both `mul` and `smul` and should represent these variations.

qadd This instruction does a signed addition and saturates the result to the signed range. We will analyze a possible variation to a normal addition.

vhsub.32 This operation represents the vector operation unit of the Cortex-A9. This includes complex instructions used for e. g. image processing. It is possible that this instruction will not be executed speculatively because of their complexity.

shift or rotate With ARMs instruction set we can perform shift or rotate operations on the last operand of instructions. We will test a shift operation exemplary with an addition to check a possible difference.

Conditional Execution ARM provides conditional execution. This means, dependent on flags of the status register instructions can be executed or not. This is possible for most of the available instructions. ARM provides this to avoid too much branch or jump overhead to increase the overall performance. We only need to append a suffix to an instruction which is then, depending on the status flag, executed or not. For illustration `addeq r1,r2,r1` is only performed if the equality flag Z is set to 1. Still, it is also potential that the multiplication itself will be executed in either case and only transferred to the output register if the condition holds. This will be discussed in more detail in Section 4.2.

Load Load instruction is possible because an array address is fetched, which requires a load instruction like `ldr r0,[r2]`[8].

Division The instruction set A32 provides no division operation [ARM16].

Listing 4.1 displays the test code for the subsequent experiments. Important is that these instructions are set and executed before the array fetch (line 4). Originally, we will only use assembly instructions because we do not want the compiler to do code optimizations. These experiments were done using the processor speed of 250 MHz.

- Additionally to a memory/array fetch, we can also execute 6 NOPs (`mov r0,r0`). Therefore the scope might increase to 41 cycles. 7 additional NOPs were not possible, indicating that the defined value was not cached.
  Unfortunately, the memory fetch might be ordered by the pipeline. So we can not be convinced that each NOP is really executed.

- Other instruction are also possible, e.g. 2 times `mov r0,#37` plus 2 times `mul r0,r1,r0` with 2 cycles per instruction. In the end, an array fetch is executed. But again, the execution might be out-of-order.

- An additional shift of the second operand had no influence on the measurement.

- Also no impact was discovered by testing the instruction `qadd r1,r2,r1`

- While we were able to perform two normal multiplications, executing a signed multiplication we could only perform one instruction additionally.

[8] Brackets tell the compiler to use the address stored in register r2

```
1 if ( x< array1_size) {
2 //Begin of speculation
3         asm volatile(
4                 "mov r1, %[s]"    "\n\t"
5                 "mov %[s],r1"     "\n\t"
6                 :[s] "+r" (x):: "memory"
7         );
8         temp = array2[x];
9 }
```

Listing 4.2: Inline assembly test: Checks whether a dependent *mov* (line 5) instruction can be executed additionally to one array fetch.

- Analyzing the conditional execution, we figured out that the condition has no effect on the speculative context.
- As expected a vector operation destroys speculation. We are not sure if the vector operation was started and not completed or that only the fact that a vector instruction is in range of speculation, the architecture does not allow out-of-order execution in this scope. We can only check whether the instruction was started or not with a power side-channel.

To avoid out-of-order execution in the speculative context we can include a dependency of the instructions to the array fetch, e.g `mov %[x],r1`[9], where we manipulate the variable x. The test code with respect to a *mov* instruction is displayed in Listing 4.2, *mul* instructions are tested in the code of Listing 4.3. The results of both tests showed that one extra instruction, with a dependency on the array fetch, is enough to prevent the array fetch. This proofs either our assumption from above, that instructions without a dependency will be re-ordered after the array fetch, or is a consequence of the pipelining mechanism of the Cortex-A9. The results of the test with regard to the speculative context are (summarized) in Table 4.1.

What about a second array fetch? Testing the code `array[array[x]]`, we can observe that just the inner fetch, `array[x]`, is cached. This means, that only one array fetch is possible. In the case of an already cached (inner) array fetch, the outer fetch has also not been executed or at least not completed. The code `array[x];array[y];` did not work either. Only the first fetch was cached speculatively. This behavior matches our expectations. Because a limitation of the speculative context is one array fetch, it is easy

[9] Inline assembly notation, which tells the compiler to use the variable x for this instruction. This value needs to be defined in the inline assembly statement as input or output value. The basic structure of arm inline assembly is `asm(comma-separated list of operations: comma-separated list of output operands: comma-separated list of output operands: clobber list to tell what registers etc. is used)`

```
1 if ( x< array1_size) {
2 //Beginn of speculation
3         asm volatile(
4                 "mov r1, #1"         "\n\t"
5                 "mul %[s],%[s],r1"   "\n\t"
6                 :[s] "+r" (x):: "memory"
7         );
8         temp = array2[x];
9 }
```

Listing 4.3: Inline assembly test, whether a dependent *mul* (line 5) instruction can be executed additionally to one array fetch.

Table 4.1: Instruction test in speculative execution. Tables show whether the instruction in column one has been executed in speculation with a particular repetition additionally to an array fetch. Column two shows the number of cycles needed according to [ARM16].

Instruction	**Cycles**	**Repetition <=**	**Speculated?**
`mov r0,r0`	1	6	Yes
`mov %[x],r1`	1	1	**No**
`mul r0,r2,r1`	2-6	2	Yes
`mul r0,r2,r1, lsl #7`	2-6	2	Yes
`muleq r0,r2,r1`	2-6	2	Yes
`mul %[x], r1, r1`	2-6	1	**No**
`muleq %[x], r1, r1`	2-6	1	**No**
`muleq %[x], r1, r1,lsl#7`	2-6	1	**No**
`add r1,r2,r1`	1	4	Yes
`addeq r1,r2,r1`	1	4	Yes
`add r1,r2,r1 lsl #2`	1	4	Yes
`add %[x], r1, r1`	1	1	**No**
`addeq %[x], r1, r1`	1	1	**No**
`addeq %[x], r1, r1, lsl #2`	1	1	**No**
`sub r1, r2,r1`	1	4	Yes
`sub %[x], r1, r1`	1	1	**No**
`orr r1, r2, r1`	1	4	Yes
`orr %[x], r2, r1`	1	1	**No**
`vhsub.32 r1, r2, r1`	n.a.	1	No
`qadd r1,r2,r1`	2	2	Yes
`smull r1,r2,r3,r4`	3 - 10	1	Yes

to explain why a second array fetch does not happen. It is also reasonable that in all three cases the amount of instruction needed for both fetches is higher than the architecture allows for speculation, and therefore the second fetch might be discarded for executing it speculatively.

Conditional Branching in Speculative Execution

The simplest side-channel attacks are made possible by branching in the control flow, for example by an `if-statement`. To investigate this possibility we used the code from Listing 4.4 with the result, that different array fetches depending on `secret` were cached. This means we can check the power consumption of different instructions. The condition itself can include sensitive information which we want to determine by checking which branch was taken. In Listing 4.4 we already gave a possible code sample in which we check whether a bit of the secret is set or not. Since it is a branch we have to analyze how this conditional jump is compiled and executed. Reviewing the assembly code shows that the if-statement is correctly compiled to a branch `beq addr`[10] and has not been optimized. To check for unexpected behavior we executed four different test run. The first one is that the variable `secret` in Listing 4.4 is always true and in the second test always false. These tests proved that the branch is executed in a speculative context. The other two tests should cover the changes from true and false to each other. Therefore, we test what happens if the secret is always true except in the speculative run and the opposite likewise. We can do that by knowing in which run we will execute the code speculatively (see Section 3.1 for explanation). Unfortunately, the last two tests revealed that the inner branch is also executed speculatively. During the training phase for the actual branch (used to enforce the speculation) we are also training the second branch.

What does that mean for us? Since we do not know the variable x and the secret in an actual use case, we do not have any information about which branch has to be rightfully executed. Maybe we are able to distinguish both branches by power consumption. Yet, we cannot be sure whether it was executed speculatively or correctly. In this context, we have maximum uncertainty. We could try to detect the rollback of internal speculation with power consumption. However, we must assume that the underlying architecture rewinds the external speculation first or both simultaneously. With a high probability, we cannot distinguish between two rollbacks. Of course, we can also train this branch, but this would destroy our attack possibility and therefore has no advantage. So we have to find a way to observe the inner speculation or conditional execution. In other words, we have to observe entries of specific prediction units, in our case the pattern history table.

[10] Branch if equal to address addr

```
1  if(x<arraySize){
2  //Start of Speculative Context
3          if(secret){
4          //Execute instruction with HIGH consumption
5                  //tmp = array[x]; used for testruns
6          }else{
7          //Execute instruction with LOW consumption
8                  //tmp = array[y]; used for testruns
9          }
10 }
```

Listing 4.4: Speculative Context Attack Possibility: Conditional Execution

This is quite difficult, we can either try a debugger or test a congruent branch to check prediction behavior. A debugger is able to freeze the architectural state of a device step by step. Our interest concerns the microarchitectural state. Whether this state can be observed depends on the debugger and on the debugging interface of the board. Using a congruent branch to discover the state of the prediction unit for the targeted branch, by testing speculation, would change the state and influence our attack.

In applications, we can assume that the `secret` is cached. The speculation happens, because of the memory latency. If the value is cached this latency is significantly lower. Maybe this will omit the speculation. But inspections showed that also with a cached `secret` the architecture speculates the inner branch as well.

Conclusion

We have tested several aspects of the speculative context. All in all, we can be sure that the boundary of the memory latency is an upper limit for the speculation. Additional instructions with an array fetch were not possible if we add a dependency between these operations. This means we can execute operations that do not take more than about 35 cycles. We found no information about which instructions can be executed or not in the related reference manuals [ARM16, XIL18, Ltd, ARM14a]. However, because the speculation has a significant impact on the performance, we assume that basic operations like multiplication can be performed in a speculative context. We also observed the speculative execution of a conditional branch. But since this branch is also executed speculatively we cannot use this execution sequence for further experiments. What we can use are all provided instructions. Some of them might need more than 35 cycles but still, start in the transient execution. This information might also be useful.

4.2 Power Consumption

In this section, we will explore the power behavior of the PYNQ-Z1. Precisely we will check the following two things:

- Has the board power consumption differences between instructions?
- Are executions of instruction with different input data distinguishable using power analysis? In other words, can we detect data-dependent power consumption in single instructions?

To measure the power consumption of the PYNQ-Z1 we will use the integrated FPGA. The software-based measuring was already described in Section 2.5. The sensor measures the power consumption indirectly and transfers the trace into BRAM. The trace has a length of 128 power values.
Unless stated otherwise, we will use the procedure described in Section 2.5 to analyze the above questions. For reliability and correctness of the measurement, we will always collect 1000 measurements for each test run. To reduce noise we will execute the tests only on one of the two cores from the Cortex-A9.

As a reminder, we will run our tests with the following procedure.

1. Collect 1000 traces of the targeted code sequences.
2. Calculate the average of the collected traces for all several code sequences. These average traces are commonly called *templates*.
3. Collect 1000 traces a second time, each randomly chosen from the targeted sequences.
4. For each new trace calculate the normalized cross-correlation to every template from step 2.
5. Choose the best correlation value and its linked code sequence and check the correctness of this detection.

For our test, we will choose the same operations as used in the previous section, where we analyzed the scope of the speculation. In addition to that, we will choose the multiplication for examining data-dependent power consumption. Power consumption may vary depending on the processed data. For example, a multiplication with zero might be handled as a special case in the responsible circuit and therefore has lower consumption than multiplication with a random value (unequal to 0). The same argument would

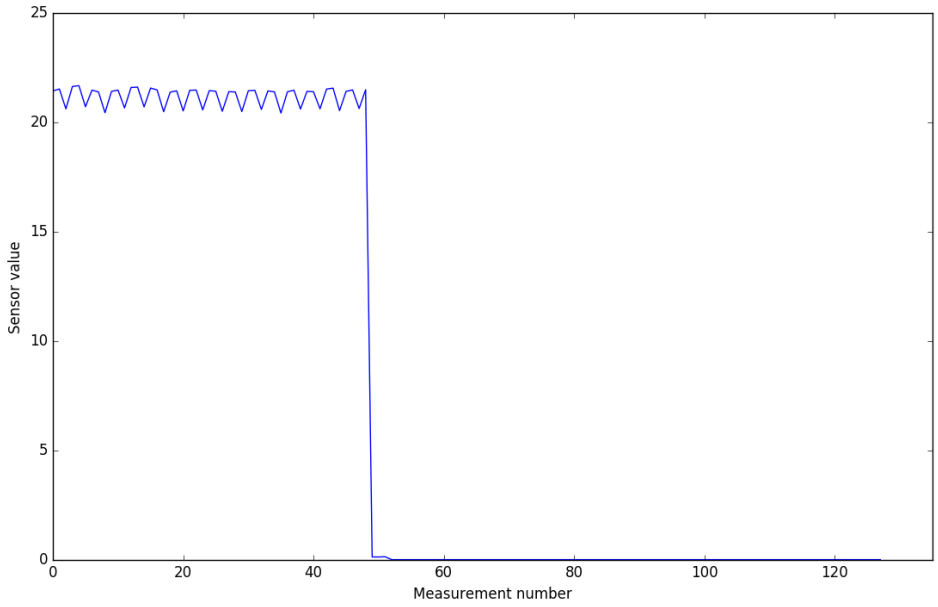


Figure 4.1: Power trace of setting and resetting the trigger. Trace shows that the operations need about 49 measurements to complete.

hold for a division. But since we cannot perform division (in assembly), we will not test data-dependent consumption of a division.

Evaluation Code Sequences

The FPGA sensor returns a power trace of length 128 (see Section 2.5). This trace might have some noise from the trigger instruction. Precisely, the sensor starts measuring while the instruction to trigger the FPGA may have not been completed on the CPU. For this purpose, we ran one time a code sequence, which only sets and resets the trigger (a GPIO pin). Afterward, we can see that both instructions needed about 49 cycles to complete (see Figure 4.1). To avoid this noise in further computations, we will calculate the normalized cross-correlation between value 49 and 128.

In Listing 4.5 we displayed an example code snippet used for testing. The structure of the test code sequences is always the same: *a)* trigger the measurement, *b)* execute test code and *c)* reset the trigger. Using the *move* and *store* operations at line four and five, we set the trigger positioned at the address of register `gp`. This register is defined inline 18 as the input parameter and refers to the `uint32_t *gpio` pointer. At line 8 we define a repetition amount of 100 using the compiler instruction `.rept`. This instruction unrolls the underly-

ing code sequence, ending with `.endr`, during compilation to a repetitive sequence. The repetition amount can be arbitrary but not a variable. The actual test operation is at line 9 and can be any instruction of the A32 instruction set used in our test board.

```
1  void snippet_NOP (uint32_t *gpio) {
2          asm volatile (
3  /* a)      Set GPIO pin to trigger the sensor*/
4                  "mov r4, #1"          "\n\t"
5                  "str r4, [%[gp]]"     "\n\t"
6
7  /* b)      Start code to evaluate from here */
8                  ".rept 100"           "\n\t"
9                  "mov r4,r4"           "\n\t"    //Replacable operation
10                 ".endr"               "\n\t"
11                 /* End code to evaluate here */
12
13 /*c)      Reset the GPIO pin*/
14                 "mov r4, #0"          "\n\t"
15                 "str r4, [%[gp]]"     "\n\t"
16 /*       Tell the compiler what input variable he should use
17         and which data is altered*/
18         :: [gp] "r" (gpio) : "r4", "memory");
19 }
```

Listing 4.5: Example code snippet used for testing power consumption. Each snippet sets the same GPIO pin to trigger the measurement and resets it at the end. Between setting and resetting the actual test code is implemented. The snippet tests the *mov* instruction, executing it 100 times (line 8).

The conditional execution is tested via the function `snippet_mul_gt` at line 10 of Listing 4.6. The code compares two registers in line 8 and if the first is greater or equal, the multiplication will be performed. In this example code, the first register holds 2 and is greater. Thus the multiplication in line 10 will be executed.

In the following, we will present our measurement results. As mentioned in Section 2.5 we will use two different sensor settings for the evaluation.

Measurements with Sensor Setup 1

The first settings use a frequency of 130 MHz for both the CPU and the FPGA. Thus, both parts run synchronously with the same speed.
Figure 4.2 displays the generated templates (average of 1000 traces) of the tested instructions. As you can examine, the templates do not differ very much. The sensor values between zero and 50 are nearly equal to one another. This is because the trigger oper-

```
1  void snippet_mul_gt(uint32_t *gpio){
2          asm volatile (
3  /* a)      Set GPIO pin to trigger the sensor*/
4                  "mov r4, #1"      "\n\t"
5                  "str r4, [%[gp]]"     "\n\t"
6  /*b) Start code to evaluate from here */
7                  "mov r0, #2"          "\n\t"
8                  "cmp r0,r4"         "\n\t"
9                  ".rept 100"         "\n\t"
10                 "mulgt r0,r0,r1"     "\n\t"
11                 ".endr"             "\n\t"
12 /*c) End code to evaluate here */
13                 "mov r4, #0"          "\n\t"
14                 "str r4, [%[gp]]"     "\n\t"
15         :: [gp] "r" (gpio) : "r4", "memory"
16         );
17 }
```

Listing 4.6: Evalutation code sequence of conditional multiplication.

ation is executed up to this point in time. After that, there are some small differences between the templates, but they are so small that you can not distinguish them well. We can confirm this by calculating the similarity between them using the cross-correlation. The correlation values are shown in Table 4.2.

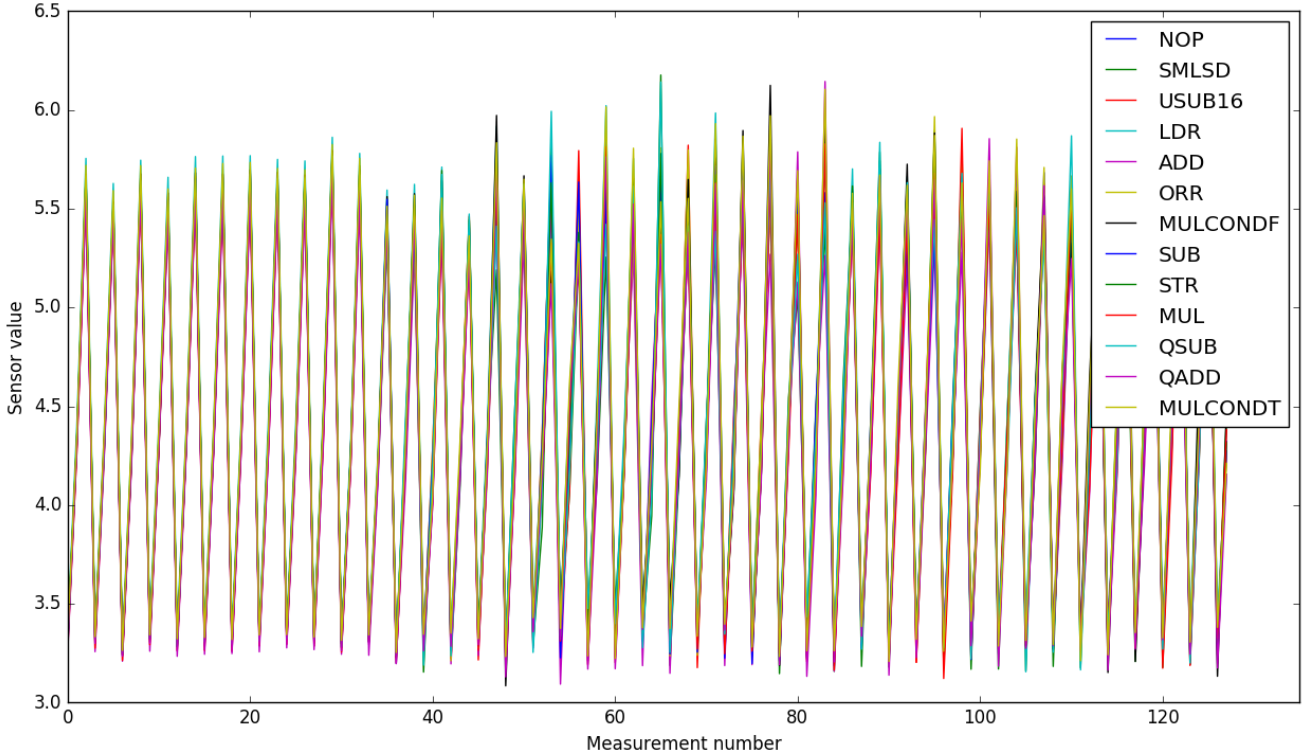


Figure 4.2: Templates of all tested instructions. The graph illustrates that the instructions do not differ much in power consumption if we use the average for each instruction.

Table 4.2: Correlation Values between generated templates using sensor setup 1

	NOP	SUB	QSUB	USUB16	ADD	QADD	ORR	SMULL	MUL	LDR	STR
NOP	1	0.9891	0.9854	0.9928	0.9845	0.9902	0.9844	0.9720	0.9891	0.9905	0.9874
SUB		1	0.9869	0.9842	0.9946	0.9908	0.9832	0.9744	0.9965	0.9929	0.9969
QSUB			1	0.9984	0.9789	0.9912	0.9923	0.9735	0.9881	0.9848	0.9828
USUB16				1	0.9750	0.9866	0.9804	0.9737	0.9860	0.9888	0.9830
ADD					1	0.9887	0.9829	0.9676	0.9945	0.9913	0.9964
QADD						1	0.9895	0.9755	0.9914	0.9959	0.9908
ORR							1	0.9739	0.9907	0.9913	0.9823
SMULL								1	0.9754	0.9790	0.9757
MUL									1	0.9948	0.9950
LDR										1	0.9932
STR											1

Table 4.3 lists the detection rate of different instructions (see Listing 4.5 for code description) using 1000 traces. We tested for each instruction the detection rate against the *nop* code sequence, by calculating the correlation and used the higher as the detection result for each run. This means, this tables shows which instructions can be distinguishable from one another. During our tests, we figured out that the trace was not completely filled with the repetition of one and therefore increased the amount to 50.
As we can see, every tested instruction can be distinguished from a *nop* sequence. Surprisingly, load and store operation were also distinguishable with a reliable probability. Conditional execution tests, on the other hand, delivered many differences between repetition amount and the instructions itself. Still, the probabilities are high enough to distinguish the conditional execution and use them for further investigations in the speculative context. We also tested *nop* vs *nop* sequences. Since these are the same instructions, we should not be able to distinguish them according to power consumption. And exactly this was proven by our measurements. The detection rate of both lies around $50\,\%$, which means the correlation values were highly similar and we can not distinguish them from another.

Data-Dependent Power Consumption

For examining data-dependent power consumption we calculated seven random values and choose two numbers ourselves. The self-chosen numbers are 0 and 2.These two might be considered separately in the architecture because multiplication by zero is always zero and multiplication by two (or any other number which is a power of two) is a logical, binary shift left. The other random values are 1369, 587, 1833, 107, 330, 1590 and 2396. These were chosen independently using the module *random* in *python*. The second operand is always the same and was also generated randomly, and has a value of 173.
The detection rates of these different values used for multiplication are shown in Table 4.4. Due to the overview and the lack of space, we show only 4 different values. All other random values have comparable detection rates. The tables show that nearly every detection rate is higher than $50\,\%$. For instance, multiplication with zero and two can be distinguished with a probability of about $70\,\%$. This value should be high enough to consider this as a possibility for further investigation. We can also see, that there are recognizable variations between the rates of different repetition amount settings. Unfortunately, the variations are distinct from measurement to measurement. This could be the result of contrasting noise levels. A third observation we can make is that the rates decrease noticeably as the number of code sequences increases. For example, when executing four different operations (see last entry of Table 4.4), the rates are just over 50 percent, which is still more than 25 percent above the expected value when we would guess the code sequence.

Table 4.3: Power measurements of different instructions using sensor setup 1, each with repetition amount 50 and 100. The data being processed is the same in every instruction. The first column describes which code sequences have been executed. The second column shows the detection rate of the sequences with different repetition amounts. The detection rate is calculated by determining how many runs of each code sequence were correctly identified. The last column shows the detection rate of the NOP instruction if it was executed.

Instruction(s)	Results in %		
	rep 50	rep 100	NOP
nop + mul	99.91	99.91	97.82
nop + add	99.97	99.71	98.72
nop + sub	99.95	99.95	98.45
nop + orr	100	99.97	99.12
nop + qadd	99.89	100	96.98
nop + qsub	100	100	97.86
nop + smull	99.97	99.95	97.56
nop + mulgt (T)	99.73	100	99.21
nop + mulgt (F)	99.85	99.75	98.87
nop + ldr	99.89	100	99.68
nop + str	100	100	100
ldr +	78.16	79.12	
str	81.96	73.25	
mulgt (T)+	74.20	74.93	
mulgt (F)	62.24	77.02	
mulgt (F− >T)+	79.22	78.41	
mulgt (T− >F)	59.33	71.16	
add +	56.64	58.73	
qadd	57.55	61.73	
sub+	68.01	38.23	
qsub+	63.22	60.96	
nop +	47.20	52.99	
nop	51.37	51.06	

Table 4.4: Power measurements of multiplication with *different* operands, each with repetition amount 50 and 100. The first column describes which code sequences have been executed. The second column shows the detection rate of the sequences with different repetition amounts. The detection rate is calculated by determining how many runs of each code sequence were correctly identified. Detection rates of *nop* sequence are not listed.

Instruction(s)	Results in %	
	rep 50	rep 100
nop + mul by 0	99.97	100
nop+ mul by 2	100	100
nop+ mul by 1369	99.97	99.93
nop+ mul by 587	99.97	100
mul by 0 +	69.98	77.19
mul by 2	65.39	77.04
mul by 0 +	79.45	67.19
mul by 1369	69.02	62.50
mul by 0 +	57.91	61.29
mul by 587	75.61	83.04
mul by 2 +	51.01	45.05
mul by 587	64.76	51.08
mul by 2 +	49.77	56.12
mul by 1369	71.62	86.68
mul by 587 +	72.19	71.64
mul by 1369	73.77	70.33
mul by 0 +	57.76	60.16
mul by 2 +	63.92	49.77
mul by 1369	75.09	32.87
mul by 0 +	68.02	75.36
mul by 2 +	62.23	47.78
mul by 587	72.00	27.07
mul by 0 +	60.52	31.79
mul by 1369 +	73.36	58.46
mul by 587	72.44	40.94
mul by 2 +	83.9	51.56
mul by 1368 +	54.08	40.03
mul by 587	42.57	67.33
mul by 0 +	57.59	67.27
mul by 2 +	39.95	46.73
mul by 1369 +	57.44	58.35
mul by 587	39.10	66.65

Combining Spectre and the FPGA Sensor

Since we want to examine the applicability of a power side-channel on Spectre attacks we want now to combine our Spectre implementation and the FPGA sensor. Unfortunately, our implementation does not work with the settings of the initial sensor setup. Because of the low frequency the range of the speculation has changed. While we reduced the speed of the FPGA and the CPU, we did not reduce the speed of the memory controller. This unit runs at $200\,\mathrm{MHz}$ and therefore is faster than the CPU. But our Spectre implementation needs the advantage of memory latency to enter the transient execution.

Due to this fact, Jonas Krautter from KIT has updated the sensor and adapted it for the speed of $250\,\mathrm{MHz}$. The memory controller remains on $200\,\mathrm{MHz}$. In the following, this will be referred to as sensor setup 2. By applying these settings, our Spectre implementation provokes again the transient execution with the same results described in Chapter 3. Nevertheless, we have to make the same measurements again to check for new results.

Measurements with Sensor Setup 2

The outcome of the measurements with the second setup is listed in Table 4.5. Like in the first setting we will, due to the trigger operation, begin calculating the correlation at tracepoint 49. Note that the table shows the results using a repetition amount of 1000. This amount ensures that the trace is filled with the defined instruction. In the former sensor setup, we used an amount of up to 100. This is also enough using setup 2 to fill the trace, but the outcomes had more fluctuations. Using the first setup we did not recognize such behavior.
The results of the second setup are different in the following aspects.

- The differentiation between *NOP*s and other operations are not as high as with setup 1. Still, the detection rates are high enough to distinguish them.

- Using this setup we can identify conditional execution (of a multiplication) with rate higher than $90\,\%$ (see (*) in Table 4.5). We can verify this property by changing the outcome of the condition in step 3 of our measurement procedure. In other words, we invert the condition in the two code snippets (`mulgt` with the fulfilled condition and with a not fulfilled one) but leaving the function addresses the same (see (**) in Table 4.5). Why do we do this? With a changed condition, we expect to get low detection rates. A new trace of e. g. the function `mulgt_TRUE()`, which refers to the case where the condition should hold (but in our current test does not hold), should correlate with the template of the opposite function `mulgt_FALSE()` significantly

Table 4.5: Power measurements with different instructions, each with repetition amount 1000. The data being processed is the same in every instruction of the left half. The right half shows multiplication with different operand. For each test, sensor setup 2 was used.

Instruction(s)	**Detection** in %
nop + mul	80.60
nop + add	74.27
nop + sub	85.77
nop + orr	81.90
nop + qadd	81.37
nop + qsub	71.46
nop + usub16	82.69
nop + mulgt (T)	73.74
nop + mulgt (F)	93.79
nop + ldr	92.87
nop + str	97.94
ldr	92.60
str	97.86
(*) mulgt (T)	93.68
mulgt (F)	93.34
(**) mulgt (F->T)	20.30
mulgt (T->F)	11,58
addgt (T)	82.56
addgt (F)	79.97
addgt (F->T)	79.51
addgt (T->F)	80.41
add	80.36
qadd	93.22
sub	73.07
qsub	90.67
usub16	68.10
nop	41.79
nop	58.09

Multiply with operand	**Detection** in %
mul by 0	76.96
mul by 2	58.29
mul by 0	70.74
mul by 1369	92.02
mul by 0	56.32
mul by 587	69.26
mul by 2	45.01
mul by 587	73.21
mul by 2	71.92
mul by 1369	86.74
mul by 587	73.23
mul by 1369	88.59
mul by 0	56.64
mul by 2	61.08
mul by 1369	90.353
mul by 0	56.67
mul by 2	55.86
mul by 587	30.91
mul by 0	69.40
mul by 1369	92.82
mul by 587	45.19
mul by 2	54.27
mul by 1369	94.93
mul by 587	50.63
mul by 0	72.78
mul by 2	11.35
mul by 1369	87.054
mul by 587	41.86

better than with the "correct" template. And like we suspected, we now get detection rates of at most $20\,\%$.
Unfortunately, we could not produce the same results for a conditional *addition* or any other arithmetic operation. The difference of power traces of the conditional multiplication is shown in Figure 4.3. The peek to peek values differ recognizably between these two templates.

- The detection rate of two *NOP* code sequences is lower than in the first setup, but still shows an appropriate sensor accuracy. The rates are shown in the last entry of the left table in Table 4.5.

- The results of the test concerning data-dependent power consumption execution differ from the first setup as well. Again, the table Table 4.5 does not show all operands being used, but contains the most expressive ones. Except for the operand 1369 all detection rates have decreased significantly. But operand 1369 has a reliable detection rate of at least $85\,\%$. But why is that only 1396 has such a high rate? If we look at the Hamming weight, we can see no reason. In addition to the random numbers generated, we also tested numbers with Hamming weights 1 to 16 and could not find any difference. Another consideration is that the address of the function in memory plays a role. But we also investigated this and could not detect any expressive effects.

4.3 Combining Spectre and the FPGA Sensor

We now want to combine our Spectre attack with the second sensor setup and investigate whether we can take advantage of the findings from the previous chapter.

Exploiting Conditional Execution in Speculative Context

We have found that we can detect the conditional execution of multiplications at a high rate. This means we can distinguish whether a multiplication was executed or not. If the condition is tied to a secret, we are able to leak it over the power consumption and reconstruct the secret. In the following, we consider the variable x as attacker-controlled data and the size of `array` to be 16. Imagine that a secret value is stored right behind `array`, so that when we access the array with a higher index we can read information about the secret. For example `array` begins at address *0x0000* and a secret starts at *0x0010*, then we can access the first bit of the secret using `array[16]`. This might not be the case in common applications but we can add an arbitrary factor to access any address in the

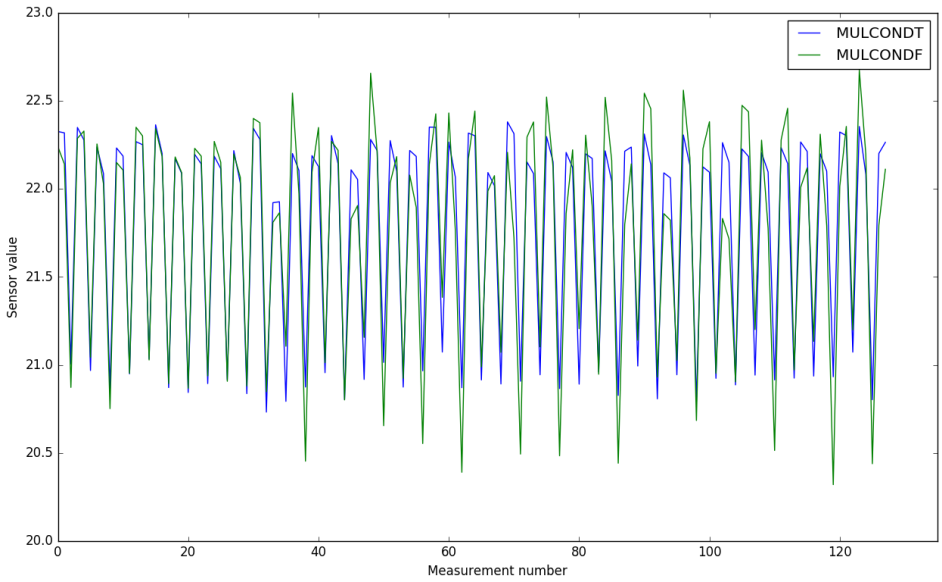


Figure 4.3: Templates of conditional multiplication differ in peek to peek values. One with the condition outcome to be true, the other to be false. X-axis: measurement index, Y-axis: sensor value

victim's address space. For simplicity, we assume that bits of the secret start right behind the last value of `array`.
Using this assumption we can create a possible attack that is displayed in Listing 4.7. This code listing shows the attack using *inline assembly* and for better understanding also a C code variant. But note that the compilation of the C code does not generate the same code in the assembly code sequence. The attempt of this attack is to leak bitwise information about the secret using the power behavior of conditional multiplication.

The procedure of this attack is shown in Figure 4.5. Again, the speculation will appear in every fifth loop (see Chapter 3 for more details). The differences toward the basic Spectre attack are marked. We had to place the trigger operation before the computation of the variable x to be greater than $arraySize$ only every fifth loop. We tried to execute the trigger operation after this computation, but this disrupted our speculation. Setting the trigger is an I/O operation which has a sufficient latency to change the program behavior. While the trigger operation is being executed, the conditional branch will be evaluated in parallel and *out-of-order*. Thus, the required memory latency of the condition is now significantly lower and therefore averts the speculative execution. Because of this behavior, we have to execute the trigger operation a little earlier.

```
1  if (x < arraySize) {
2  /* In speculation x has a value of 170
3    For an attack this value can be considered as secret*/
4          asm volatile(
5                  "mov r0, %[s]"  "\n\t"
6                  "mov r1, #1"    "\n\t"
7                  "cmp r0,r1"    "\n\t"
8                  ".rept 100"   "\n\t"
9                  "muleq r0,r1,r0" "\n\t"
10                 ".endr"          "\n\t"
11                 "mov %[s], r0"  "\n\t"
12                 :[s] "+r" (array[x]):: "memory"
13         );
14 }
```

(a) Assembly Code of attack possibility

```
1  if (x < arraySize) {
2  /* In speculation x has a value of 170
3    For an attack this value can be considered as secret*/
4          if(array[x]==170){
5                  array[x]*=170;
6          }
7  }
```

(b) C Code of attack possibility

Listing 4.7: An Attack possibility in which one wants to take advantage of different power consumption of conditional execution of a multiplication. a) shows the attack in assembly and b) in C code.

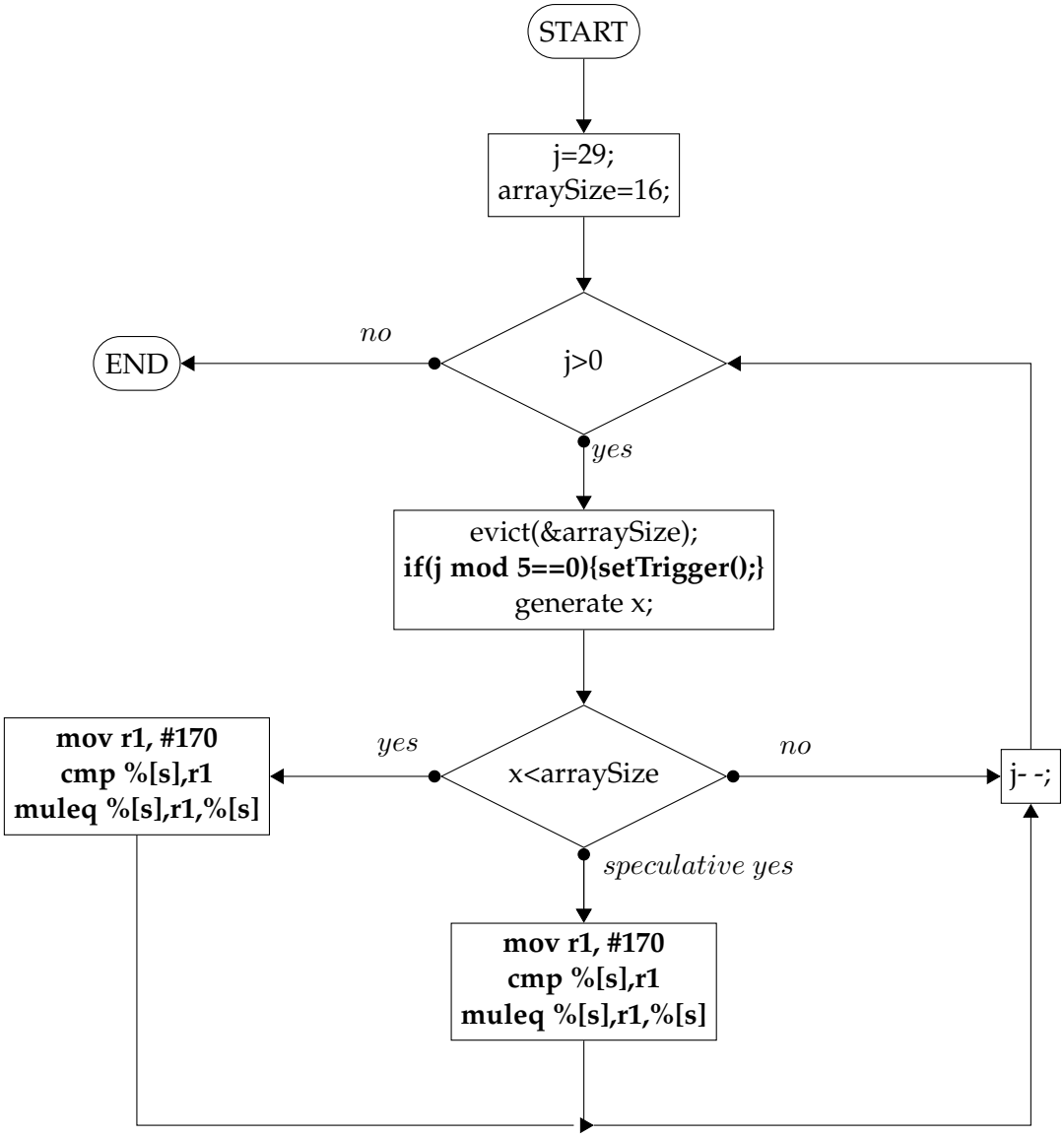


Figure 4.5: Attack Flow of the first attack possiblity - Exploiting Conditional Execution in Speculative Context. In every fifth loop the program will enter the speculative path.

Power Measurement Results

Let us analyze the power trace of the Spectre code including conditional multiplication in the transient execution. Again, we calculated the average of 1000 traces and use them as templates. The two templates, one with a positive condition and the other with a negative, are shown in Figure 4.6. In this graph, we can see some differences between the templates. We can see some differences concerning the peek to peek values, but it is quite difficult to observe. Therefore, let us compare the *spectre* templates with the *conditional multiplication* templates for both cases, where the condition is fulfilled and not fulfilled. These two cases are shown in Figure 4.7. In both cases, we can recognize significant differences in the traces. What does that mean?
We can assume that the Spectre code generates more noise than the measurement of the multiplication itself. So, we will use all four templates to calculate the detection rate. But first, we have to check the content of the traces, at which trace index the speculation starts. As aforementioned we have to set the trigger with a little distance to the conditional branch. This means the trace contains additional computation and the speculative context may start in a later part of the trace. We determined the start with the same technique as described in Section 4.2. The start point is at trace index 80 and our correlation will also start at this point. But note, that it is possible that the start/stop overhead is mainly caused by stopping and the speculation might already start at index 40.

The differences in the templates can also be recognized by calculating the correlation. The correlation between a new *raw-spectre* trace and each template are on average lower than 0.35, which is a low value. As a reminder, this value measures the similarity between two traces and the maximum similarity has a correlation value of one. The detection rate of 1000 traces using the four templates are listed in Table 4.6. We can remark, that the detection rate of templates with the positive condition is higher in both the correct and incorrect cases. Thus, we can not be sure if the bit of the secret is set and therefore a multiplication was performed. Our detection will always predict a multiplication, and the bit to be 1 respectively. In other words, we have too many *false positives*.

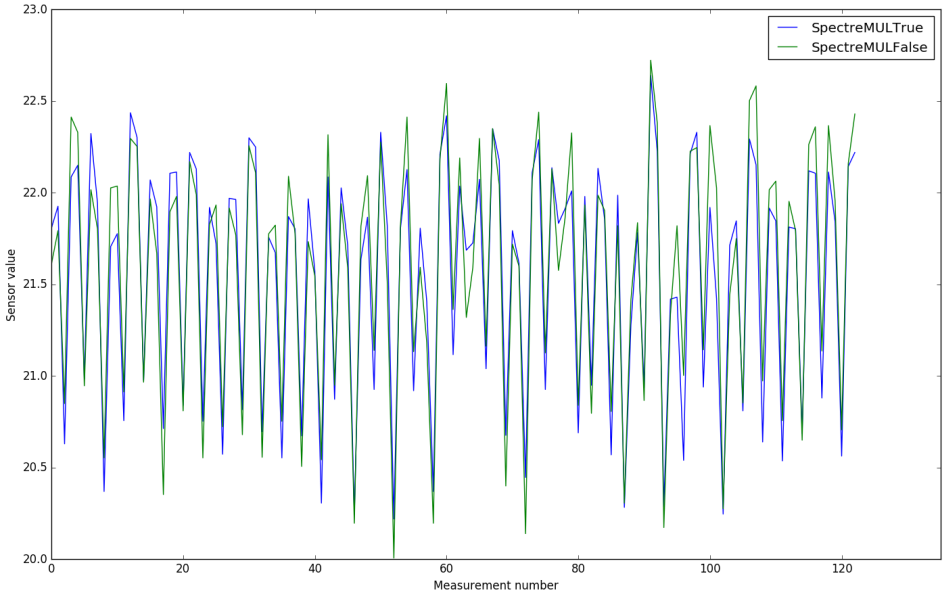
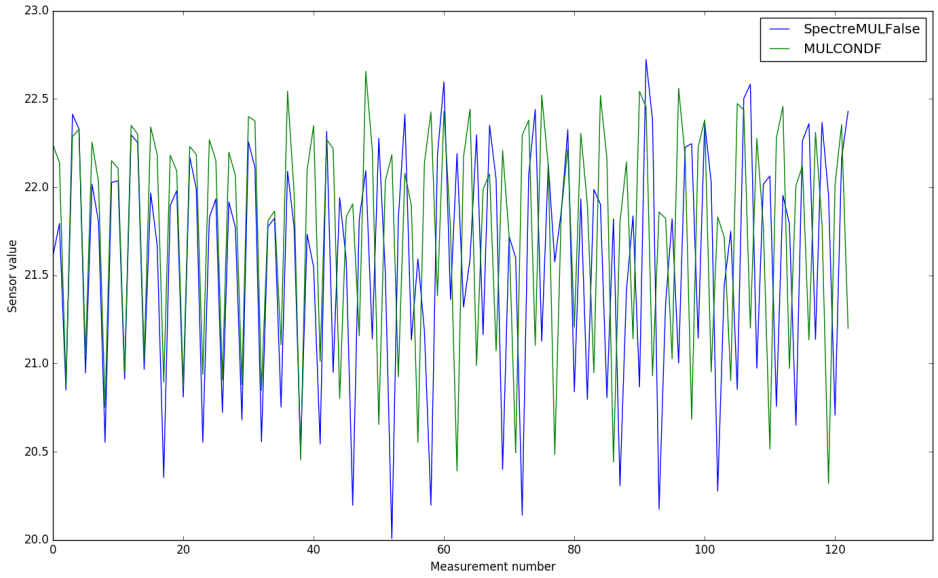


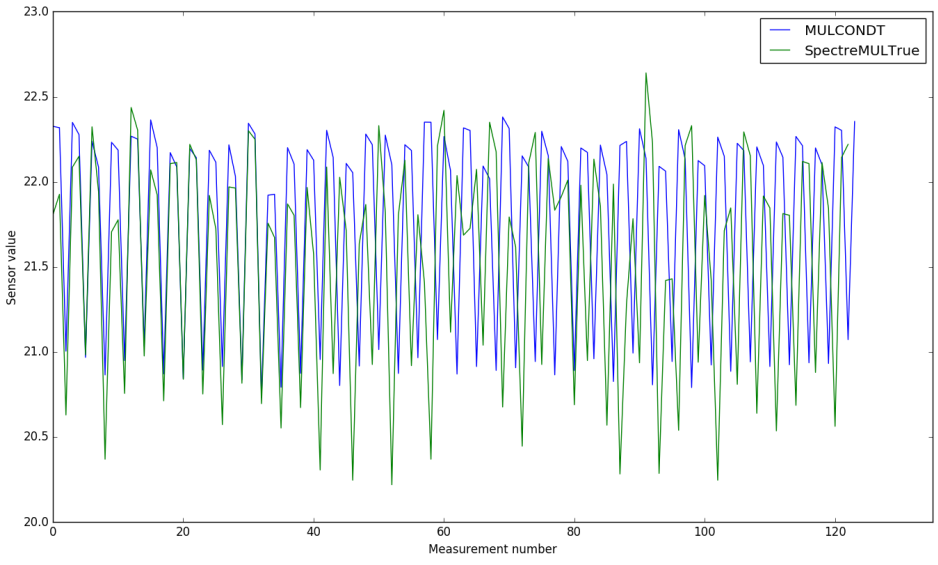
Figure 4.6: Spectre with the attack to exploit conditional execution. One template in which the conditions holds and one in which it does not.

Table 4.6: Detection rates in the percentage of 1000 Spectre traces using four generated templates presenting the average Spectre trace and average multiplication trace depending on the outcome of the conditional execution.

Template	Condition is True	Condition is False
SpectreMulgt T	67%	64%
SpectreMulgt F	33%	36%
Mulgt T	58%	56%
Mulgt F	42%	44%



(a) Templates of conditional execution in spectre environment and in standalone environment, each where the condition is wrong and the multiplication should not be performed.



(b) Templates of conditional execution in spectre environment and in standalone environment, each where the condition is true and the multiplication should be performed.

Figure 4.7: Templates of conditional multiplication in two environments where in (a) the condition is not fulfilled and (b) the condition is fulfilled.

5 Conclusion and Discussion

This master thesis investigated the combination of Spectre attacks and power as the leaking covert channel. Previous attacks use the *cache* as the necessary leaking channel.
We first showed that the Spectre attacks *Spectre-PHT* and *Spectre-BTB* are possible on the used ARM-platform *PYNQ-Z1* by implementing them as described in Chapter 3. Our research on these attacks illustrates that for a proof of concept that the latter has no advantage to *Spectre-PHT*. Instead, *Spectre-PHT* is easier to implement and easier to understand.
Based on a quantitative and qualitative analysis of the power consumption of independent instructions and data-dependent computations, it can be concluded that it is possible to use power traces to identify specific code sequences. Thus, power consumption can leak sensitive information on the targeted ARM platform.
However, our investigation of the combination of our Spectre implementation and the FPGA sensor showed, that we no longer have the accuracy to detect specific code sequences. This means we could not prove the suitability of a power side channel to leak information from the transient execution. As there is a time limit for the master thesis, this is our final result. Nonetheless, on the basis of these results, we can be confident that the combination of power analysis and Spectre attacks is still achievable if further optimizations are done. Some promising optimizations are discussed in the next section.

Discussion

During our examinations, we continuously had ideas for improvements. Those which we have not been investigated in this master thesis are explained in the following.

- Our Spectre implementation takes advantage of memory latency. So that the processor does not have to wait inactively for the response of memory access, it executes subsequent operations transiently. To execute operations transiently (or speculatively this memory latency is necessary and inevitable. Perhaps, one can try to use *I/O* latency to improve the time in the speculative context and offer new leakage possibilities. The latency of *I/O* operation might be larger than memory access.

- To facilitate attacks more realistically, future research should also consider running Spectre-BTB using a *congruent* branch for the training phase. This would expand a proof of concept to a realistic scenario in which the attacker is able to run a process

on a system with the same or different address space. However, it is required to reverse engineer the prediction unit to find a congruent branch. The advantage is to force the victim to execute arbitrary code. In our implementation, we show a proof of concept and assume the victim to execute targeted code sequences.

- Calculate the power templates of each instruction with the cross-correlation. Power traces of the same code sequence can differ in an arbitrary shift. Identifiable features can be in one trace at trace index 12 and due to a shift in a second trace at index 46. Thus, calculating the average may polish the identifiable features of the traces in such a way that these will not appear in the template. The cross-correlation shifts one trace above the other and calculates the similarity after each shift. We could then use the (n-times) shifted trace with the best correlation to calculate the overall average. Using this approach we might generate a more unique and expressive template of a specific code sequence so that we possibly increase the accuracy and the detection rate.

- Another aspect that future work has to consider is that the power measurements from the transient execution may not be contained in the expected trace area. It is likewise possible that the speculative part is relatively short and therefore may already be over by the time current measurements are taken. Therefore, further investigation is needed to locate the targeted code in the trace. One can also try to use a larger trace to extend the possibilities.

- In Section 4.2 we also explored data-dependent power consumption. Especially the value of 1369 was distinguishable with a high detection rate. So we promising area which needs further experiments.
  A possible attack variant is in Listing 5.1. This attack multiplies the secret either with zero or 1369. In line 4 we store the last bit of the secret in register `r1`. The secret is stored in `array[x]` defined in line 8. Just like in Section 4.3 we assume that `x` is attacker-controlled. Then we multiply this bit with 1369 (line 6). The resulting operand contains now either zero or 1369, depending on the secret bit. The multiplication we want to leak via a power trace is in line 7. This multiplication can be repeated to improve the measurement results.

```
1  if (x < arraySize) {
2  /* In speculation x has a value of 170*/
3      asm volatile(
4          "and r1, %[secret], #1"  "\n\t"
5          "mov r3, #1369"         "\n\t"
6          "mul r1,r1,r3"         "\n\t"
7          "mul r2,r1,r3"          "\n\t"
8          :[secret] "+r" (array[x]):: "memory"
9      );
10 }
```

Listing 5.1: Attack possibility to take advantage of data-dependent power consumption of a multiplication. This code performs a multiplication if the secret bit (stored in `array[x]`) is 1.

References

[ABuH+] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *Port Contention for Fun and Profit*, page 0. IEEE.

[AL17] Na-Young Ahn and Dong Hoon Lee. Countermeasure against side-channel attack in shared memory of trustzone. *CoRR*, abs/1705.08279, 2017.

[ARM14a] ARM. *ARM Architecture Reference Manual*, armv7-a and armv7-r edition, 5 2014.

[ARM14b] ARM. *Cortex-A-series Programmers Guide*, 4.0 edition, 1 2014.

[ARM16] ARM. *ARM Cortex-A9 Reference Manual*, 2 2016.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Ber] Daniel J Bernstein. Cache-timing attacks on aes.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO*, 1997.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. *CoRR*, abs/1903.01843, 2019.

[CBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on a.e.s. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied*

Cryptography and Network Security, pages 293–306, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[DMWS12] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 106–117, June 2012.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[GOKT16] D. R. E. Gnad, F. Oboril, S. Kiamehr, and M. B. Tahoori. Analysis of transient voltage fluctuations in fpgas. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 12–19, Dec 2016.

[GRLZ+17] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, Vancouver, BC, 2017. USENIX Association.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[Hor18] Jann Horn. Speculative Execution, variant 4: Speculative store bypass. 2018.

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

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptol-*

ogy Conference on Advances in Cryptology, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[KW18] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[Lew01] J.P. Lewis. Fast normalized cross-correlation. *Ind. Light Magic*, 10, 10 2001.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association.

[LLLGCN16] Ruben Lumbiarres-Lopez, Mariano Lopez-Garcia, and Enrique Canto-Navarro. A new countermeasure against side-channel attacks based on hardware-software co-design. *Microprocessors and Microsystems*, 45:324 – 338, 2016.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[Ltd] ARM Ltd. Speculative processor vulnerability. `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability`. Accessed: 2019.10-05.

[Man03] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology — ICISC 2002*, pages 343–358, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[SGMT18] Falk Schellenberg, Dennis Gnad, Amir Moradi, and Mehdi Tahoori. An inside job: Remote power analysis attacks on fpgas. pages 1111–1116, 03 2018.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[SSL+19] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 279–299, Cham, 2019. Springer International Publishing.

[SSLG18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[VBMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[VCMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 740–757, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Wil12] Barry Duane Williamson. Line allocation in multi-threaded hirarchical data stores. Patent US8271733 B2, Arm Ltd., 2012.

[WMW19] Jack Wampler, Ian Martiny, and Eric Wustrow. Exspectre: Hiding malware in speculative execution. In *NDSS*, 2019.

[XIL18] XILINX. *ZYNQ-7000-SoC*, 1.12.2 edition, 7 2018.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[ZSZF13] Kenneth Zick, Meeta Srivastav, Wei Zhang, and Matthew French. Sensing nanosecond-scale voltage attacks and natural transients in fpgas. pages 101–104, 02 2013.