Detecting Microarchitectural Attacks

DRAFT

Author: *Magdalena M. Solitro*

Contents

1 Introduction **1**

2 Static analysis for cybersecurity **3**
- 2.1 Theoretical foundations . . . 4
- 2.2 Practical approaches and related challenges . . . 6
  - 2.2.1 Taint analysis . . . 6
  - 2.2.2 Symbolic execution . . . 8

3 Spectre **11**
- 3.1 Background . . . 11
  - 3.1.1 Out-of-Order execution and micro-operations . . . 12
  - 3.1.2 Speculative execution . . . 12
  - 3.1.3 Cache side channels . . . 13
- 3.2 Spectre variants . . . 14
  - 3.2.1 Spectre-PHT (Pattern History Table) . . . 15
  - 3.2.2 Spectre-BTB (Branch Target Buffer) . . . 16
  - 3.2.3 Spectre-RSB (Return Stack Buffer) . . . 17
  - 3.2.4 Spectre-STL (Store To Load) . . . 18

4 Verification tools to detect Spectre **21**
- 4.1 FastSpec . . . 21
- 4.2 KLEESpectre . . . 23
- 4.3 Testing the Advanced Encryption Standard (AES) . . . 27
  - 4.3.1 Testing with KLEESpectre . . . 27
  - 4.3.2 Testing with FastSpec . . . 31
  - 4.3.3 Consideration about the results . . . 31

5 Conclusions **33**

A Deep Learning **35**
- A.1 Generative Adversarial Networks (GANs) . . . 35
- A.2 Bidirectional Encoder Representations from Transformers (BERT) . . . 36
  - A.2.1 Transformers . . . 37
  - A.2.2 Architecture . . . 39

A.2.3 Training 39

B Block ciphers and AES **41**
B.1 Block ciphers 41
B.2 Modes of operation 42
B.2.1 Cipher Block Chaining (CBC) mode 42
B.2.2 Output Feedback (OFB) mode 43
B.2.3 Counter (CTR) mode 43
B.3 AES: Advanced Encryption Standard 44

Chapter 1

Introduction

Within the harware products market, companies such as Intel, AMD, Nvidia, and so on, strive to produce faster, smaller, energy-efficient chips to beat the competition and grant themselves the biggest market share. However, achieving the best results in every aspect is concretely infeasible: for instance, highly performing devices tend to consume a greater amount of power, while chips must trade small dimensions with memory capacity. Therefore, the main challenge is to reach an acceptable trade-off between all the different aspects that influence the performance of a device.

Speed is a hot topic when it comes to CPU design: most users want their computer to go *fast*, and to satisfy this request, engineers keep coming up with all sorts of expedients. One of the most effective dodges to achieve higher performances is *speculative execution* (also known as *transient execution*), which consists in retrieving in advance from the main memory instructions that have a higher chance to be executed, to reduce the time that the processor spends idle, waiting for the instruction and the data involved to be loaded in the cache. This comes useful in presence of conditional branches: since it's impossible to statically infer what branch is going to be selected, the CPU uses statistical information about previous executions to *speculate* about how the conditional expression will be evaluated, and thus what set of instruction will need to be retrieved.

The introduction of speculative execution successfully improved the CPU performances, but at the same time it introduced some unforeseen security flaws that were discovered only after these processors had already been installed in thousands of machines: Flush+Reload [43], Spectre [19], Meltdown [22], Spoiler [15] are the names of some vulnerabilities that exploit a chache side-channel opened by transient execution to expose potentially secret information.

Starting from 2018, the year in which these attacks were disclosed, many efforts have been made to patch them, but ditching speculative execution completely was never an option, as the impact on performances would have been too great. Instead, a series of strategies have been engineered to at least mitigate its consequences on the security of programs. Besides this, another problem needed to be addressed was how to figure out whether a software was vulnerable or not to such side-channel attacks. One way to do it is to use *static analysis* to inspect the code before its execution and detect specific patterns or possible execution paths that can lead to a data leakage. While these techniques are not guaranteed to always provide the correct result, they are considered to be highly reliable to verify the properties of a program.

In this thesis, we target a specific security vulnerability, Spectre, and we use static analysis techniques to inspect popular software libraries to verify if they can suffer from Spectre-like attacks. We chose specific libraries from OpenSSL[1], a project that aims to provide robust, general-purpose software to ensure secure communication over the network. We decided to test OpenSSL for many reasons: it's open-source, it's written in a language that lend itself well for the analysis that we plan to carry out, and it's one of the most widely used libraries at the moment. Furthermore, OpenSSL notoriously suffered from various security bugs since its earliest versions (one of the most serious ones was Heartbleed[2], disclosed in April 2014), so I thought it would be interesting to see if it's a fertile ground for attacks based on speculative execution. More precisely, in this thesis we submit selected OpenSSL libraries to two different static analysis tools, that employ different detection strategies: one dates back to 2020 and is considered to be a state-of-the-art tool, while the other one is more recent and uses an uncommon, yet promising approach that was not seen in any other analysis tool for Spectre detection. At the end of our study, we will not only test whether OpenSSL is immune or not to Spectre-like vulnerabilities, but we will also compare these two tools from different points of view, to evaluate their effectiveness and their performances.

The thesis is structured as follows: Chapter 2 covers the main notions about static analysis and its theoretical basis, along with a focus on some practical strategies employed by one of the tools that we will use later; Chapter 3 is instead focused on Spectre and will present not only the microarchitectural details that allow to exploit the side effects of speculative execution, but will also provide an in-depth description of all the Spectre variants currently known. We then proceed with Chapter 4, that contains the core of this work: it describes the tools selected to test OpenSSL, providing an overview of the strategies they enforce to detect Spectre and presenting the experimental results obtained by submitting the selected libraries to these tools. The rest of the thesis is dedicated to the appendices, where we cover notions on complexity theory, deep learning, and cryptography that can come useful to those readers who are not familiar with the topics for comprehending various part of this thesis.

---

[1] https://www.openssl.org/

[2] https://heartbleed.com/

Chapter 2

Static analysis for cybersecurity

Static analysis is a set of automated techniques used to inspect a program to check whether it satisfies certain properties, without executing it. The origins of static analysis date back to the Seventies and it was initially developed to speed up the compilation process. Nowadays, these techniques are used for a multitude of purposes, such as program verification, synthesis of optimised code, and certification of critical software applications [42].

This chapter will provide an insight on the theoretical principles and practical approaches of static analysis, with particular attention to the cybersecurity perspective. Some of the question concerning safety and security that we can tackle with static analysis are, for example: will the software crash under certain circumstances? Will specific inputs give illicit access to information that shouldn't be disclosed? Does the software accomplish *exactly* what it is programmed for or will there be undesired side effects? We can try to give an answer to these questions through intensive testing, but this approach is destined to be a coarse approximation of the correct answer: the domain of possible inputs that need to be examined is often far too broad to be tested in a reasonable amount of time and selecting a small subset of inputs is likely to leave out those few that can cause a malfunctioning. In addition to that, nowadays computer programs are behind the correct functioning of safety-critical systems, such as industrial machinery or avionics: in these contexts, where unexpected behaviours can likely cause serious injuries or even death, the analysis of the program cannot be left to inaccurate methods. These are just a couple of reasons that highlight the necessity of techniques that allow us to verify quickly and precisely the properties of a program.

We begin our dive into static analysis by firstly providing the theoretical arguments on which this field is based and then we will move on to describe some modern strategies to carry out the analysis.

2.1 Theoretical foundations

The goals of static analysis are ambitious, because it aims to provide tools that can deliver the following advantages:

- **Speed:** the possibility of using methods to analyse code quickly implies being able to perform more controls, more frequently;
- **Automation:** being able to check the properties of a program automatically relieve humans from a cumbersome and error-prone process;
- **Precision:** static analysis can detect flaws that appear only very rarely during execution, and thus would be hard to spot through simple testing.

These objectives can be partially achieved, although not entirely: static analysis is not infallible, which means that it does not always provide the correct results. To be more precise, the effectiveness of static analysis has to confront some intrinsic limits of computation, namely the undecidability of the *Halting problem* and *Rice's theorem.*

Theorem 1 (Halting problem)**.** *Let $\mathcal{L}$ be the set of all programs that can be written in a certain language, and let p be one of such programs.*

The halting problem consists in finding an algorithm `halt` *such that,*

$$\forall\, p \in \mathcal{L} \,.\, \texttt{halt(p)} = \textit{true} \iff p \textit{ terminates}$$

The halting problem is not computable: an algorithm such as `halt` *does not exist, as proved simultaneously by Alonso Church [7] and Alan Turing [38] in 1936.*

This theorem provide an example of something that a program analysis tool cannot detect: no algorithm can decide whether a program, written in any language, will terminate or not.

There is another theoretical result, that provides an even stronger statement on what we can prove about the properties of a program:

Theorem 2 (Rice's Theorem)**.** *Let $\mathcal{L}$ be a Turing-complete language, and let $\mathcal{P}$ be a nontrivial semantic property of programs of $\mathcal{L}$. There exists no algorithm* `SatP` *such that,*

$$\forall\, p \in \mathcal{L} \,.\, \texttt{SatP(p)} = \textit{true} \iff p \textit{ satisfies the semantic property } \mathcal{P}.$$

The notion of "nontrivial" mentioned in the theorem identifies those properties that either concern *every* program in the language, or *none*, therefore

$$\mathcal{P} \text{ is trivial} \iff \mathcal{P} = \mathcal{L} \lor \mathcal{P} = \emptyset$$

Clearly, we are not interested in trivial properties, because there is really nothing to prove about them in a program! Intuitively, what Rice's theorem states is that, for *any interesting property*, we cannot have an algorithm that is able to decide whether a certain program has that property or not. This sounds discouraging: the theorems mentioned above seem to destroy any hope of being able to prove anything interesting about programs. Luckily, this is not quite the case: while it is impossible to have an automated procedure that correctly detects a certain property in *all* cases, it is perfectly possible that it does it *sometimes*. This also means that it's impossible to construct a "perfect" static analysis tool, namely one that is always able to detect any (extensional) property of a program: this justifies the existance of a research field dedicated to static analysis.

What we just said implies that the results provided by the algorithm will occasionally produce incorrect results, such as false positives and false negatives: the former refers to the detection of bugs that the program doesn't actually have, while the second one regards the failure of finding bugs that the code indeed has. False negatives are clearly a much bigger issue, because they lead to a false sense of security: for this reason, a good tool for static analysis should never fail to detect bugs, while it is allowed to output a false positive [12]. Another way around the limitation of Rice's Theorem is to give up complete automation, designing tools that require human intervention to compute the final result, accepting the risk of introducing mistakes in the computation of the final result.

The desiderable properties of a program analysis tools can be formalised by means of two notions: *soundness* and *completeness*.

Definition 1 (Soundness)**.** Let `analyse` be a program that tests whether another program has a certain property $\mathcal{P}$ and let $\mathcal{L}$ be a programming language. We say that the program `analyse` is **sound** with respect to $\mathcal{P}$ if the following condition is satisfied:

$$\forall p \in \mathcal{L} \,.\, \texttt{analyse}(p) = \textbf{true} \implies p \text{ satisfies } \mathcal{P}$$

In simple terms, a proof system is sound if all the statements it proves are actually true. A trivial `analysis` program that is guaranteed to be sound is the one that always returns false: by invalidating the premise, it makes the overall implication to be true, even though such analyser would clearly be of no utility.

For the definition of completeness, we stick to the same notation.

Definition 2 (Completeness)**.** We say that a program `analyse` is *complete* with respect to $\mathcal{P}$ if the following condition is satisfied:

$$\forall p \in \mathcal{L} \,.\, p \text{ satisfies } \mathcal{P} \implies \texttt{analyse}(p) = \textbf{true}$$

As the definition suggests, a *complete* analyser is one that is able to prove *all* the properties that are satisfied by a program. More generally, a formal system is said to be complete with respect to a particular property, if every formula that satisfies that property is a theorem (i.e. can be proved) within the system.

Also in this case, we can provide a trivial `analysis` program that is guaranteed to be complete, namely the one that always returns true. In fact, if the consequence is true, the overall implication will always evaluate to true, independently from the truth value of the premise.

Soundness and completeness are two essential aspect of a correct analysis, but an inherent limit of computability prevents us from having both. This limit is a consquence of Gödel's Incompleteness Theorems, which essentially show that there is an intrinsic gap in mathematics between *proof* and *truth*. These theorems are extremely complex and profound, and thus we will not dive into their details, but the interested readers can find out more about the topic in [31], [36], [14].

2.2 Practical approaches and related challenges

This section discusses the general concepts behind the approaches used by the tools discussed in Chapter 4. The description should be considered as a high-level introduction whose aim is to give an idea of the mechanisms leveraged by these tools, but more accurate descriptions can be found in the papers cited throughout this section.

2.2.1 Taint analysis

Taint analysis is a technique that involves labelling each variable supplied to the program by an untrusted source as *tainted*, assuming that the untrusted source could be an adversary trying to attack the system through that variable. Then, every variable in the program whose value is dependent from tainted variables is also marked as tainted. In this way, we can keep track of the propagation of tainted data in the following instructions and detect when this data is used in dangerous ways. Taint analysis can be used to identify parts of code vulnerable to Spectre: for instance, in a Spectre-PHT attack, certain values can be supplied repeatedly to a conditional branch to induce a misprediction. The following example shows a Spectre-PHT gadget where the tainted variables are marked in red:

```
1 int checkValue(int x){
2     if(x < len(array1)){
3         y = array2[array1[x] * 4096;
4     }
5     return y;
6 }
```

The parameter `x` is supplied by the user, therefore is marked as tainted. The value of `y`, determined in line 3, depends on `x`, thus `y` becomes automatically tainted. This is an example of taint (dynamic) analysis *based on data flow*, because it involves marking external taint data and tracking its propagation throughout the execution.

This type of analysis can be used for vulnerability detection [29], malware analysis [4] [44], and web applications [3] [30], and can be applied both statically or dynamically.

Dynamic taint analysis can be based either on *data flow* (the example above is an instance of this approach) or on *control flow*. The latter involves constructing a control flow graph (CFG) of the program, by examining the branching instructions that govern the execution of the code [8].

Challenges in taint analysis

The main issues that taint analysis encounters are **under-tainting**, **over-tainting** and **state space explosion**. Let's start with the first one: under-tainting means that the analysis tool does not mark as tainted a data that was should be marked as such. This happens when the data is not arithmetically derived by an untrusted input or, more generally, is not linked to such input by an explicit instruction in the program, but is still influenced by it. This phenomenon causes the detection of **false negatives**, namely the tool recognises a piece of code as secure, even though it's not. A clever example of how this can occur is proposed in [29]:

```
if(x==0){
    y=0;
} else if (x==1){
    y=1;
} else if (x==2){
    y=2;
} ...
```

This conditional structure goes on in the form

```
if(x == n){
    y = n;
}
```

The program is semantically equivalent to the following instruction:

```
y = x;
```

A taint analysis will mark `y` as tainted in the last instruction, but not in the first example.

The opposite problem is **over-tainting**, which leads to the detection of false positives: a piece of code can be evaluated as insecure, even in the absence of any feasible execution that taints a certain data. As already stated previously in this chapter, false positives are preferred to false negatives, since the latter can give a false sense of security and fail to notify potential vulnerabilities. One source of over-tainting can be a conservative extraction of the Control Flow Graph (CFG), which is defined as follows:

Definition 3 (Control Flow Graph (CFG))**.** A *control flow graph (CFG)* is a directed graph $G = (V, E)$ in which nodes represent the program's instructions, while an edge $u \to v$ represents a possible flow from the instruction u to the instruction v. This means that there exists at least one run of the program in which the execution of u is followed immediately by the execution of v. The set V of nodes contains also two special nodes:

START, that has no predecessors and from which every node is reachable, and *END*, which has no successors and is reachable from every node.

The construction of an accurate graph from the binary is often challenging, due to the difficulty in determining the targets of indirect branches and function calls. For this reason, tools such as oo7 [41] use a conservative approximation of the CFG, which can contain more edges that expected.

The last issue worth mentioning is the **state space explosion** (also known as path explosion), which can occur during the construction of the CFG or the Control Dependency Graph (CDG). The latter is defined as follows:

Definition 4 (Control Dependency Graph (CDG))**.** A *control dependency graph (CDG)* [21] *is a directed graph* $G = (V, E)$ *in which nodes represent the program's statements or expressions, while an edge* $x \to y$ *expresses the fact that the statement* x *assigns a variable which is used in* y*, which means that the mere execution of* y *depends on the value of the expression* x*.*

The CDG is used by the analyser to keep track of how the tainted data affects the other variables in the program. However, as the complexity of the program increases, the size of these graphs grows exponentially and can become infeasible to analyse. This issue is also encountered in another popular approach, symbolic execution, which is discussed in the below.

2.2.2 Symbolic execution

One way to test the behaviour of a program is to provide it with different, random inputs and observe the result of every execution. However, this approach presents a glaring limit: the domain of all the possible inputs can be extremely wide, if not infinite, which makes it infeasible to try all of them. One way around this obstacle is to use symbolic execution [2] [17]: instead of using fully specified input values, we simultaneously explore multiple paths that a program can take under different inputs. This is possible by representing those inputs that cannot be determined statically under a symbol, namely an abstract value. Such inputs could be, for example, paramenters that must be provided by the user at run time.

During the symbolic execution of a program, the analysis engine keeps track the following information:

- The next *statement* to evaluate (`stmt`).
- A *symbolic store* σ, that maps the variables of the program either with expressions over concrete values or with symbolic values.
- Some *path constraints* π, namely "a formula that expresses a set of assumptions on the symbols due to branches taken in the execution to reach `stmt`" [2].

Different execution paths distinguish themselves by the assumptions that are made about the symbolic values. To give an idea about how this process works, consider the following piece of code:

```
1 int func(int a){
2     int x = 1;
3     bool y = true;
4     if (a != 0){
5         ...
6     }
7 }
```

At the beginning of the symbolic execution, the state maintained by the engine will be:

$$(\texttt{stmt}, \sigma, \pi) = (\texttt{int x = 1}, \sigma = \{a \mapsto \alpha_a\}, \pi = \mathit{true})$$

As you can see, `a` is a paramenter that can be known only at run time, as it is a user-supplied parameter. Therefore, the symbolic store will assign it to the symbolic value α_a: this is a way of stating that, according to our current knowledge, `a` can assume any value in the range of integers, but we don't make any assumptions about it because it's not necessary (yet). π is initially set to *true* because we are currently making no assumption about the value of any symbol.

The next step represent what happens after the (abstract) execution of `int x = 1`. The state is modified as follows, where the coloured elements are those that differ from the previous step:

$$(\texttt{stmt}, \sigma, \pi) = (\texttt{bool y = true}, \sigma = \{a \mapsto \alpha_a \, x \mapsto 1\}, \pi = \mathit{true})$$

Once the instruction on line 2 is executed, the symbolic store can map the variable `x` with the *concrete* value 1, which can be inferred statically.

The next instruction involves a branch conditioned on the value of `a`, which is not known. Therefore, the symbolic execution engine produces two paths, one for $\alpha_a = 0$ and the other for $\alpha_a \neq 0$:

$$(\texttt{stmt}, \sigma, \pi) = (\texttt{if (a != 0)}, \sigma = \{a \mapsto \alpha_a \, x \mapsto 1\}, \pi = \{\alpha_a = 0\})$$

$$(\texttt{stmt}, \sigma, \pi) = (\texttt{if (a != 0)}, \sigma = \{a \mapsto \alpha_a \, x \mapsto 1\}, \pi = \{\alpha_a \neq 0\})$$

Through this kind of branching, the symbolic execution evolves as a tree until the program has terminated or until we gained enough knowledge to determine whether the property of interest is satisfied or not.

Challenges in symbolic execution

The tree structure of a symbolic execution suggests that this kind of analysis suffers of the same issue that we have already encoutered with taint analysis, namely **path**

explosion (or state space explosion): as the size of the program grows, the number of feasible paths grows exponentially, and can even be infinite if the program contains unbounded loop iterations. Although an exhaustive exploration of the state space is the only way to guarantee a sound and complete analysis, a partial exploration is sufficient in many scenarios to prove a certain property, thus the problem of path explosion can often be circumvented.

Another problem of symbolic execution concerns **constraint solving**. As precisely stated in [2], "in a symbolic executor, constraint solving plays a crucial role in checking the feasibility of a path, generating assignments to symbolic variables, and verifying assertions". These constraints are expressed in a logical language and their truth value is evaluated by decision procedures called "constraint solvers". Finding a solution for such formulas is notoriously an NP-complete problem: for this reason, one of the most popular approaches to constraint solving consists in mapping the formula to an instance of the boolean satisfiability problem (also known as SAT, see Appendix **??**), a famous NP-complete problem for which many efficient algorithms have been developed. For those problems that cannot be easily mapped to SAT, we can use Satisfiability Modulo Theories (SMT), that generalize the SAT problem with supporting theories to capture more complex formulas. Even though some significant advances has been accomplished in recent years to optimise the search of a solution, complexity remains a major obstacle to the scalability of symbolic execution to large programs. Moreover, not even SMT solvers are powerful enough to handle constraints that involve certain operations, such as non-linear arithmetic, which prevents us from exploiting the advantages of optimised SMT solvers.

Chapter 3

Spectre

On January 2018, two major vulnerabilities affecting the vast majority of modern CPUs were disclosed, taking the names of *Spectre* [19] and *Meltdown* [22]. These two attacks were independently discovered by various researchers and were published in two works, [19] and [22], destined to leave a permanent mark in the field of cybersecurity. More precisely, [19] identified *Spectre*, a family of attacks that leverage on an optimisation technique used in modern processors to read and write protected memory from a program's address space, while [22] describes *Meltdown*, an attack able to bypass the privilege checks that usually prevent a user process to access regions of memory that are under the control of the operating system. Both are classified as *cache side-channel attacks* and were found to affect Intel, ARM and AMD processors, present in the vast majority of desktops, laptops, cloud servers, and even smartphones[1].

This chapter is focused on Spectre, which is classified as an *access-driven cache side-channel attack*, as it is based on the adversary's ability to monitor cache accesses made by the victim and measuring the time difference between a cache access and a memory access.

This chapter provides all the background notions that are needed to comprehend the mechanism of a Spectre attacks. Furthermore, it contains a complete overview of all the variants of the attacks that are currently known, describing what microarchitectural data structure they exploit and how the attack is carried out.

3.1 Background

This section deals with all the background concepts that lie at the basis of Spectre, in particular speculative execution and cache side channels. As stated in the pioneering work by Kocher et al. [19], "Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels."

---

[1] Source: `https://spectreattack.com/`

3.1.1 Out-of-Order execution and micro-operations

When executing a program, the processor splits single assembly instructions in a series of lower-level operations called *micro-operations* (also abbreviated as micro-ops or μ-ops): this has the advantage of allowing the CPU to execute different part of the instructions in different moments, based on the current availability of the data [11]. For example, consider the following piece of code, written in Intel syntax:

```
1 add eax, ebx
2 add [reg], eax
```

The instruction on line 1 adds the content of two cache registers, `eax` and `ebx`. This means that the data will be immediately available to the CPU and thus this instruction doesn't need to be split in more than one μ-op. The instruction on line 2, instead, is much different: it requires to (i) retrieve `[reg]`, namely the data located at the memory address stored in `reg`, then (ii) add it to the content of `eax`, and finally (iii) to write the result back to `[reg]`. Therefore, up to two memory accesses are needed to complete the instruction, but since these accesses are very time-consuming, the CPU splits the instructions into three μ-ops (corresponding to the three step described), so that it can execute other tasks while it waits for the data to be accessible, avoiding to waste hundreds of clock cycles and thus speeding up the computation. This is precisely what is referred to as *out-of-order execution*. This paradigm, however, comes with a challenge: when executing μ-ops out of order, the CPU must establish whether there are dependencies between different instructions that can obstacle the completion of certain tasks. For example, let's consider again the two assembly instructions written above: note that the addition on line 2 involves `eax`, which is modified by the previous addition. This means that the micro-operation (ii) can't be executed before the result of the previous addition is not stored in `eax`. To implement out-of-order execution successfully, the CPU uses a mechanism called *memory disambiguation*, accurately described in 3.2.4, to detect and resolve this kind of dependencies.

3.1.2 Speculative execution

Speculative execution is an optimization technique where the CPU uses statistical information about the program execution to make guesses about the outcome of conditional branches or a data dependencies, and consequently decides to load data and execute certain micro-operations in advance. If the prediction is correct, the processor can benefit of the intermediate results and the data that loaded to speed up the execution of the following instructions. Otherwise, the CPU performs a rollback to the last correct state, discarding the intermediate results computed speculatively. It's imporatant to highlight that the effects of speculative execution are visible only at microarchitectural level until they are committed to the architectural state: the user doesn't have the perception that the execution flow is not strictly linear, and never sees the effect of incorrect predictions in the program's output. However, when a misprediction happen, the intermediate results and the data loaded preventively in cache remain there. This asymmetry between the microarchitectural state and what the user sees from the execution of the program gives space to the a type of attack vector known as *cache side channel*, which is discussed in 3.1.3.

3.1.3 Cache side channels

A cache side channel [45] is a type of attack vector that allows to infer secret variables by monitoring the state of the cache during the program execution. So far, three different kinds of cache side-channels have been identifies, each of which monitors different behaviours of the cache: time-driven [33], trace-driven [33], and access-driven.

Time-driven side channels In this case, the adversary keeps track of the execution time of the program and use that information to infer what data is loaded into the cache. A typical example of timing attack targets the modular exponentiation function used in many public key cryptographic algorithm, included RSA. Modular exponentiation consists in raising a large number (i.e. the plaintext or the ciphertext) to a large exponent (i.e. the public key or the secret key respectively), which is extremely time consuming, since it requires to perform repeated multiplications using large integers. The subprocedure `square-and-multiply`, allows to do this computation efficiently:

```
square-and-multiply(M, e, N){
    R = 1
    for (i=0 to n-1){
        R = R^2 mod N
        if(e[i] == 1){ R = R * M mod N }
    }
    return R
}
```

where `M` is the message we want to encrypt (or decrypt), `e` is the exponent, `n` is the its length, `N` is the modulo, `R` is the result. As one can see, for each bit of the secret or public key `e`, the message is squared. If the current bit of the exponent is 1, then an additional multiplication is performed on the result: this means that, when the value of `e[i]` is 1, the computation on `R` takes longer to complete. This time difference could potentially be exploited to figure out which bits are set to 1 in the key, even though most systems nowadays employ some countermeasures to prevent it (for example, by adopting constant-time algorithms).

Trace-driven side channels In this attack, the adversary monitors the amount of power consumed during the execution. The power trace is a rich source of information, as it can reveal not only when certain operations are performed, but also what data is being used at each stage of the computation. The attacks carried out through this vector are also known as *power analysis* attacks and they proved to be very effective: for example, numerous attacks of this kind have been successfully conducted on AES, the standard algorithm for simmetric encryption, to exfiltrate entire bytes of the symmetric key [23] [25] [26] [32]. The attack is performed collecting a large number of power traces, on which the adversary performs different statistical analyses that can reveal the data dependency between the power consumption and the execution time. This attack style is known as *Differential* Power Analysis (DPA), to distinguish it from *Simple* Power Analysis (SPA), where the attacker needs only one single trace and tries to deduce information about the secret key from it.

Access-driven side channels Access-driven attacks, like time-driven ones, leverage on time measurements to unveil the value of some secret information. However, they rely on a much finer kind of measurement: while time-driven attacks are based on the execution time of the whole program, access-driven ones exploit "the ability to detect whether a cache line has been evicted, or not, as the primary mechanism for mounting an attack" [28]. From this point of view, this attack requires a much finer measurement capacity. A famous example of how to concretely exploit the timing information is FLUSH+RELOAD [43], an attack that relies on sharing pages between the attacker and the victim processes. FLUSH+RELOAD can be summarised through the following steps: the adversary's goal is to verify whether the victim uses a secret piece of data d. To accomplish this, the adversary's process evicts the cache line containing d and waits a certain amount of time, to give the victim the opportunity of using d, in case they need it. Then the attacker reloads the memory line and measures the time to load it: if the victim accessed d during the latency time, the reload will be very quick, because d was already brought back to the cache. Otherwise, d is still in the main memory and the reloading time will be significantly longer, indicating that the victim did not access d during the latency. Figure 3.1, taken from the paper that first described the attack [43], can help clarifying these concepts. One of the assumptions that must hold to carry out such attacks successfully is that the adversary must share the cache space with the victim.

3.2 Spectre variants

As mentioned at the beginning of this chapter, the term "Spectre" on itself does not identify a single attack, but a *family* of attacks, all united by the same common denominator: they maliciously influence the speculative execution of a program to exfiltrate confidential information. Different version of Spectre can be carried out through the exploitation of different microarchitectural behaviours. The purpose of this section is to present the four main variants of Spectre, describing what vulnerability they take advantage of and in how they differ from each other.

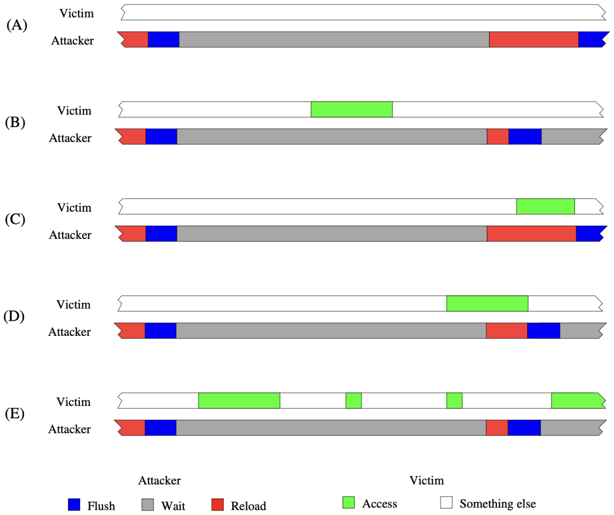


Figure 3.1: Flush+Reload time measurement [43]. (A) and (B) represent the main cases described above. (C), (D), (E) are more peculiar cases.

3.2.1 Spectre-PHT (Pattern History Table)

Spectre-PHT [19] [6] [10], also known as variant 1 or Bound Check Bypass, was one the first attack of the Spectre family to be discovered. As the name suggests, this version targets the Pattern History Table (PHT) to trigger a branch misprection.

Pattern History Table (PHT) The Pattern History Table [11] is a data structure used by the branch predictor to guess the outcome of a conditional branch before the value of the guard is fully determined. Intuitively, during the execution of a program, the CPU keeps track of the last N outcomes of a conditional branch in the *branch history register*. The outcomes are encoded as a bitstring of length N, where a 1 indicates an execution where the guard expression evaluated to true, while a 0 represents the opposite case. The content of the branch history buffer is used to point at a specific entry of the Pattern History Table. The PHT contains 2^N entries, i.e. as many entries as all the possible sequences of N outcomes; each entry in the contains a 2-bit saturating counter, namely a finite-state automaton that decides the probability of a certain outcome. The branch history register is used for choosing which of the four counters to use.

The attack By poisoning the PHT, the adversary can induce the branch predictor to make a wrong guess about the direction of the conditional branch and perform both reading and writing operations in memory parts they shouldn't have the right to access. To clarify this concept, I propose a couple of examples, taken from [6]:

```
if(x < len(array1)) {
```

```
2     y = array2[array1[x] * 4096]
3 }
```

Note that the index we use to access `array2` is multiplied by 4096. The reason is that the usual cache block size is 64 bytes, so by using indexes in the form `[k * 4096]` we avoid having two elements used in the program falling into the same cache block. The conditional instruction allows reading a value from `array2` only if `x` represents a valid index. However, after repeatedly executing the conditional branch with a value of `x` that satisfies the guard, the branch predictor will forecast that the expression `x < len(array1)` evaluates to true and thus will execute the assignment on line 2 in advance. When the adversary supplies an invalid value for `x`, the CPU will performe an *out-of-bounds memory access*, loading in the cache a value for `y` that shouldn't be accessed.

The same effect can be exploited to perform a write operation on the array, as shown in the following piece of code:

```
if (x < len(array)) {
    array[x] = value;
}
```

Just as the described above, the attacker can "train" the CPU to guess a certain outcome of the branching instruction and then exploits the misprection caused by the provision of an invalid value of `x` to write on a protected location.

3.2.2 Spectre-BTB (Branch Target Buffer)

Spectre-BTB [19] [6], also known as variant 2 or Branch Target Injection, affects the Branch Target Buffer to deflect the transient execution to a mispredicted branch target. This attack was discovered simultaneously to variant 1 and exploits a similar mechanism.

Branch Target Buffer (BTB) The Branch Target Buffer [34] is a data structure located in the cache, that stores a set of guesses for the target addresses of all jumps, both conditional and unconditional. The first time a jump instruction is executed, the target address that is reached gets stored in the BTB, so no speculation is made for the first jump. When the jump is executed again, a pointer to the BTB indicates the target address of the previous execution, allowing the CPU to fetch the predicted instruction into the pipeline, but the true target will not be calculated until the jump reaches the execution stage. Once the jump is actually performed,the address predicted with the BTB is compared with the actual address taken by the jump: if they don't match, the guess was wrong, so the results are rolled back and the previous target address is replaced by the current one in the BTB.

The attack The logic that governs this variant is very akin to the one behind Spectre-PHT: both attacks target data structures that aim to anticipate the outcome of jump instructions, trying to influence the prediction about what is executed as a consequence of the jump.

However, there is a significant difference between Spectre-PHT and Spectre-BTB: in the former, the execution flows along a restricted mispredicted path, i.e. the attacker can influence the branch predictor only in the choice of the branch to execute. The latter, instead, allows redirecting the control flow to an arbitrary destination, so that the execution can continue at a specific point chosen by the adversary. This location corresponds to a *Spectre gadget*, namely "a code fragment whose speculative execution will transfer the victim's sensitive information into a covert channel" [19].

3.2.3 Spectre-RSB (Return Stack Buffer)

Spectre-RSB [24] [20] [6], also known as ret2spec, is a variant that exploits a data structure called Return Stack Buffer.

Return Stack Buffer (RSB) The Return Stack Buffer is a microarchitectural buffer used to predict return address of a function: whenever a call instruction is reached during the execution, the prediction of the return address is pushed on top of a stack. When the execution reaches the `return` point, the top entry of the stack is used to speculate about the return address location quickly. Meanwhile, the actual return address is fetched, possibly from the main memory, therefore it will be available after only after hundreds of clock cycles. Once the real return address is loaded, it gets compared with the address that was fetched from the RSB: if they match, all the result computed until that point can be committed and the overall execution time gains in speed.

The attack To describe Spectre-RSB attacks, we also refer to another kind of stack: the *program stack*, namely a data structure that stores information about the active subroutines of a computer program.

There are various ways in which the adversary can poison the RSB, all described in detail in [20]. One way to carry out the attack is to exploit the stack overfill or underfill: the RSB has a very limited size, which can vary between 4 and 24 entries, thus it saturates quickly. When this happens, the stack gets updated in a cyclic manner, namely the latest return address is pushed on the stack, the n-th entry is discarded and the $(n-1)$-th takes its place. As the functions progressively reach the `return` instruction, the entries of both the program stack and the RSB get popped, but at some point we reach the function whose value has been overwritten, causing an underfill of the RSB.

The primary attack strategy is to directly pollute the RSB: the adversary can overwrite the return address on the program stack, so that the top entry represents the return address of the previous function. In this way, the address found on the program stack and the on top of the RSB will certainly mismatch.

Another way is to leverage on the *speculative* pollution of the RSB. When functions are called during a speculative execution and a misspeculation happens, their results are rolled back and they are removed from the stack. However, the guessed return addresses remain in the RSB, providing the opportunity to push an address that points outside the address space accessible by the program without raising exceptions.

3.2.4 Spectre-STL (Store To Load)

Spectre-STL [6], also known as variant 4 or Speculative Store Bypass, is a variant that exploits not only dependencies in the control flow, but also those in in the *data flow*. More precisely, this version takes advantage of the memory disambiguation mechanism that is put in place by most moder processors.

Memory disambiguation Memory disambiguation[2] is a set of techniques used to execute memory access instructions out of order, without affecting the value of the final result. To justify the need of these techinques, I will present two examples, both taken from (Wikipedia):

```
1 add $1, $2, $3    # R1 <= R2 + R3
2 add $5, $1, $4    # R5 <= R1 + R4
```

The code above shows two micro-operations that perform simple additions. The result of the second instruction depends on the result of the first one, since the value of register R1 is computed in line 1 and then R1 is used as operand for the addition in line 2. This is a case of *static* dependence, because the sources and destinations of the operations are registers. The processor can easily spot this dependence and decide an order of execution where the first addition is performed before the second one, so that the result is consistent with a sequential execution.

However, "complications arise when the dependence is not statically determinable". Consider the following code snippet:

```
store $1, 2($2)    # Mem[R2+2] <= R1
load  $3, 4($4)    # R3 <= Mem[R4+4]
```

In this case, the location of the operand is indirectly specified by means of a register, rather than directly defined in the instruction encoding itself. As clearly stated here, "the microprocessor cannot statically determine, prior to execution, if the memory locations specified in these two instructions are different, or are the same location, because the locations depend on the values in R2 and R4". As a consequence, it is not possible to determine at compile time whether these two instructions can be executed in a different order or not: this is known as *ambiguous* dependence. Detecting and resolving ambiguous dependencies require more sophisticated techniques than static dependency, and this is where the memory disambiguation mechanism comes into play.

---

[2] https://en.wikipedia.org/wiki/Memory_disambiguation

To further improve the performances, some processor support a technique called **memory dependence prediction**, which is analogous to branch prediction, that leaves room for a Spectre attack. This method aims at predicting the true dependencies between store and loads, in order to speculatively execute certain memory accesses out of order without affecting the final result of the computation. Similarly to the other variants, the guesses of the memory dependence predictor must be validated or discarded later in the pipeline, when the memory disambiguation system takes action and determines whether the loads and store where correctly executed.

The attack Spectre-STL exploits the prediction mechanisms to read a value from a protected address. The core idea of the attack leverages on the saw called *RAW hazard*: "RAW" stands for "Read-After-Write" and it indicates a data dependency where a load operation reads a value from a memory location that was subjected to a store operation in a previous instruction. A RAW-hazard takes place when the processor reads the address *before* the previous store operation commits its value to the memory.

The following example clarifies how to exploit a RAW-hazard to carry out a Spectre attack:

```
1 ptr = secret_pointer;     // initial value
2 ptr = public_pointer;     // STORE
3 if(is_public(ptr)){
4     value = *ptr;         // LOAD
5 }
6 cache = array[value];     // look-up
```

On line 1, the value of the pointer gets initialised with the location of a secret value, while on line 2 it gets re-assigned to the location of a public value. The `if` branch checks whether the pointer corresponds to a public address, in which case it allows to read the value stored there. The attack succeeds if the adversary can induce the CPU to delay the commitment of the second store operation, executing the `if` branch with the initial value of `ptr`.

Note that this variant does not involve manipulating the branch predictor in any way: throughout the attack, the `if` clause always evaluates to `true` (explain better).

Chapter 4

Verification tools to detect Spectre

Formal verification consists in checking whether a program implements a given specification, i.e. the program is functionally correct. *Formal analysis*, on the other hand, is used to verify whether the code satisfies certain properties, e.g. specific security guarantees. In our case, the security property that we want to target is the vulnerability against Spectre-like attacks.

This work aims at comparing two different tools: on one hand we have FastSpec [37], a recent tool that uses deep learning techniques not only to detect Spectre, but also to generate new gadgets using a GAN-based framework. On the other hand we have KLEESpectre [40], a tool that employs symbolic execution extended with cache modelling and speculative execution to detect Spectre vulnerabilities in the code. The core of this work consists in testing the security of symmetric ciphers in OpenSSL[1], a popular software library used to ensure secure communication over the network.

As specified in [37], FastSpec was already used to test OpenSSL v3.0.0 on public-key algorithms and procedures for digital signatures, namely RSA, EDCSA and DSA. Our work targets one specific symmetric-key cipher, AES (Advanced Encryption Standard, see Appendix B), in different modes of operation.

The rest of the chapter is organised as follows: Section 4.1 and 4.2 provide an overview on FastSpec and KLEESpectre respectively, describing their main features and analysis strategies. Then we proceed with Section 4.3, where we present the experimental results obtained by testing the OpenSSL functions both with FastSpec and KLEESpectre, and we conclude with Section ...., where we make some final considerations about the results we obtained.

4.1 FastSpec

As mentioned at the beginning of the chapter, FastSpec was proposed in 2021 in [37] and so far it represents the only analysis tool that leverges on deep learning to detect Spectre. All the essential notions on deep learning that are needed to understand how the tool works can be found in Appendix A. The GitHub repository of this tool can be found at `https://github.com/vernamlab/FastSpec`.

---

[1] Official website: https://www.openssl.org/

In the paper, the authors present two main ideas:

1. They propose two different methods to generate a large dataset of Spectre gadgets from the 15 originally published in [18]: the first one is a traditional algorithm based on mutational fuzzing, while the second one, called SpectreGAN, leverages on a deep learning framework named Generative Adversarial Networks (GANs) [13].

2. They describe FastSpec, that uses both on GANs and on Natural Language Processing techniques (see Appendix A) to detect Spectre vulnerabilities.

In this section, we will focus only FastSpec, as the other algorithms are out of the scope of this thesis.

FastSpec is essentially a lightweight version of BERT: as described in the dedicated appendix, BERT is a neural network that learns language, by gaining understanding of its syntactical structures and comprehending where different words fit in different contexts. FastSpec is inspired by the same idea, but instead of working with texts written in some human language, it applies it to texts written in a *machine* langauge (Assembly, in this case). The tool should be able to learn the grammatic of low-level code and what syntactical pattern can lead to a Spectre vulnerability.

The training of FastSpec is identical to a classical BERT network, thus requires a phase of pre-training followed by fine-tuning (see Appendix A).

Pre-training To "teach" Assembly to the network, we train it in two unsupervised tasks, the Masked Language Model (MLM) and Next Sentence Prediction (NSP). The former requires us to "mask" random tokens in a piece of code and make the network "guess" their value based on contextual information. The latter, instead, is more complex: normally, it would consist in providing two sentences to the network, that are temporally related to each other, and make the network learn what sentence comes logically before the other. However, Assembly doesn't have syntactical separations as periods or paragraphs: for this reason, Assembly functions are split in pieces of maximum 50 tokens each, and for every piece of function, the following segment is assigned to the label `IsNext`, while a random piece of function is given with label `NotNext`.

At the end of the training, the FastSpec should have assigned an embedding vector to each token, according to the meaning it learned during the pre-training phase. Tokens with a similar "meaning" (for example, jump instructions or register names) should be mapped to vectors that are close to each other. The authors proved the success of the mapping by depicting a map of the vector distribution in a 3-D space, that we report in Figure 4.1. Note that the vectors live in a high dimensional space, which is impossible for us to visualise: for this reason, the authors employed a specific algorithm (t-SND) to map n-dimensional vectors (where $n = 64$ in this case) into a 3-D space.

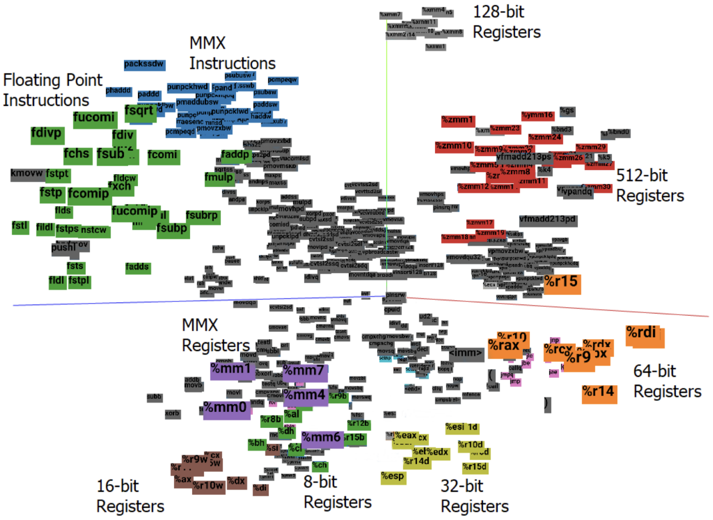


Figure 4.1: 3-D visualisation of the embedding vectors learned by FastSpec: similar instructions and registers are grouped together by the same color, while unrelated instructions and registers are separated in plane by the axis (source: [37])

Fine-tuning In this case, the fine-tuning phase corresponds to a supervised sequence classification: FastSpec needs to learn the conceptual differences between Spectre gadgets and general-purpose functions through labeled pieces. More precisely, the former are labeled with a 0, while the latter are labeled with a 1, and FastSpec is trained to distinguish between the two.

Concretely, FastSpec is trained on a dataset made by a combination of gadgets generated wiht SpectreGAN and disassebled Linux libraries, for a total of 107 million lines of Assembly code. Some minor modifications are made to the code to reduce the vocabulary size: for example, similar tokens are merged together, while all labels, immediate values and out-of-vocabulary tokens are replaced with `<label>`, `<imm>` and `<UNK>`, respectively.

The evaluation of the tool gave promising results: as the paper reports, they "obtained 1.3 million true positives and 110 false positives (99.9% precision rate) in the test dataset, demonstrating the high performance of FastSpec" and they "only have 55 false negatives (99.9% recall rate)".

4.2 KLEESpectre

KLEESpectre [41] leverages on symbolic execution (see Section 3.1.2) and modelling of cache accesses to perform the analysis of a program. As the name suggests, this tool was obtained by modifying the symbolic execution engine KLEE [5] to model the branch speculation that are the root of Spectre vulnerabilities. The GitHub repository of this tool can be found at `https://github.com/winter2020/kleespectre`.

Speculative symbolic execution As already discussed in Section 2.2.2, standard symbolic execution analyses code by representing the different paths that the program can take and abstracting the unknown inputs with symbolic values. For example, consider the following piece of code shown below. A standard symbolic execution engine like KLEE would explore the paths shown in Figure 4.2: since `x` is a user-supplied value, and therefore cannot be determined statically, the KLEE would model the two possible outcome for the first conditional statement, on line 5: either the branch condition on `x` is true (p_{T1}) or it is false (p_{F1}). The same reasoning is made for the second conditional statement, on line 8.

```
1  uint32_t SIZE = 16;
2  uint8_t array1[16], array2[256*64], array3[16];
3  uint8_t foo(uint32_t x) {
4      uint8_t temp = 0;
5      if (x < SIZE) {        // branch b1
6          temp = array1[x];
7          temp |= array2[temp];
8          if (x <= 8) {      // branch b2
9              temp |= array2[8];
10         }
11     }
12     temp |= array3[8];
13     return temp;
14 }
```

KLEESpectre, instead, makes a step further, considering four different scenarios for each branching instruction. For instance, the execution of the first branch is modelled as follows:

1. p_{T1}: `x < SIZE` is satisfiable and the branch `b1` is correctly predicted. In this case, the symbolic execution will fork a new state with constraint `x < SIZE` and proceeds by executing the code fragment highlighted in blue.
2. p_{F1}: `x >= SIZE` is satisfiable and the branch `b1` is correctly predicted. In this case, the symbolic execution will fork a new state with constraint `x >= SIZE` and proceeds by executing the code fragment highlighted in pink.

These are the two cases that are also considered by a standard symbolic execution engine, to which two more are added:

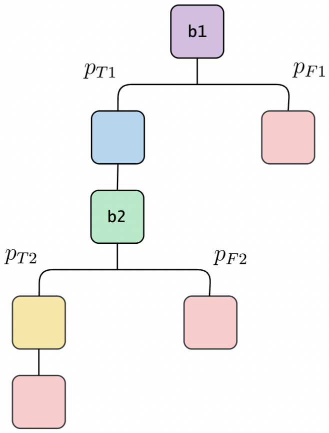


Figure 4.2: Standard symbolic execution of the program.

3. sp_{T1}: `x >= SIZE` is satisfiable and the branch `b1` is mispredicted. In this case, KLEESPECTRE forks a new state with constraint `x >= SIZE`, but proceeds by executing the code fragment highlighted in blue.
4. sp_{F1}: `x < SIZE` is satisfiable and the branch `b1` is mispredicted. KLEESPECTRE forks a new state with constraint `x < SIZE`, but proceeds by executing the code fragment highlighted in pink.

The decision tree produced by KLEESPECTRE as depicted in Figure 4.3

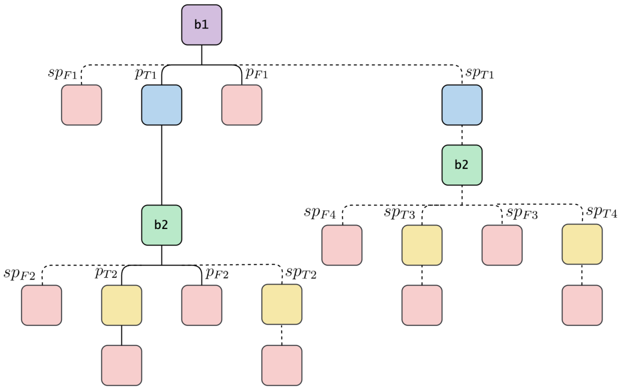


Figure 4.3: Speculative symbolic execution of the program.

The critical scenario, namely the one that poses a security threat, is the third one: the processor executes the instructions contained in the scope of the branching statement, even though the condition is not satisfied.

KLEESPECTRE handles the problem of state space explosion by discarding any path that do not pose any risk of data leakage. In the example shown in Figure 4.3 the symbolic states sp_{T3},sp_{F3}, sp_{T4} and sp_{F4} are all discarded once KLEESPECTRE reaches the limit of speculative execution window (SEW), namely the maximum number of instruction that are executed speculatively before the outcome of the branch is resolved.

Cache modelling When it comes to detect Spectre vulnerabilities, being able to determine the instructions that can be executed speculatively is not enough: such instructions are "dangerous" only if they load sensitive values in cache. For this reason, speculative symbolic execution is further extended with *cache modelling*, which allows to keep track of the values stored in cache during the speculative execution by each instruction in the path. When the execution terminates, KLEESpectre computed a symbolic cache model $\Gamma_{spectre}$, namely a set of all memory accesses, that allow to reconstruct what data is still in cache and is thus in danger of being leaked.

KLEESpectre distinguishes between two types of memory access:

- *concrete* accesses, made to a known memory address;
- *symbolic* accesses, made to an address that can be determined only at run time, and thus must be abstracted with a symbolic value.

The cache model $\Gamma_{spectre}$ is said to be satisfiable if a possible secret s, accessed along a speculative path, remains in the cache after the program execution. The size of the generated cache model depends on the Speculative Execution Window (SEW), namely the maximum number of instruction that can be speculatively executed before the outcome of the branch is determined.

Implementation KLEESpectre is built upon the symbolic execution engine KLEE [5] and takes in input LLVM bitcode, generated from the original C program using LLVM 6.0. The three main changes that KLEESpectre imprements with respect to KLEE are the following:

1. Generation of speculative symbolic states;
2. Propagation of potentially sensitive data;
3. Symbolic modelling of cache behaviour;

As described in Section 2.2.2, the symbolic execution engine maintain a state at each step of the execution, where it keeps track of certain information necessary to proceed correctly, such as the next statement to execute, a symbolic store, and path constraints. KLEESpectre generates two extra symbolic states (one of these is the cache state) to model speculative execution and takes them in consideration in the path selection heuristic (i.e. an heuristic that determines what "direction" is going to be firstly developed by the engine).

Furthermore, KLEESpectre leverages on taint analysis to track the propagation of sensitive data. As stated in the paper:

> "When a memory load instruction reads a variable v_s from an out-of-bound memory location, we mark v_s as sensitive. All new expressions dependent by v_s are subsequently marked as sensitive as well." (source: [40])

This mechanism is described more in depth in Section 2.2.1.

Finally, KLEESpectre models the cache to check whether a sensitive address can be observed by a potential adversary at the termination of a program. As quickly mentioned above, the execution state contains also the cache state, to record what information is stored in cache at each step of the execution.

4.3 Testing the Advanced Encryption Standard (AES)

This section is the core of the thesis: here you will find a precise definition of the research question that I decided to tackle, a description of the methodology that I followed, and the discussion concerning the results I obtained.

As I previously mentioned at the beginning of this chapter, the goal of my work is to test the OpenSSL library that implements AES with different tools and verify the presence of vulnerabilities. More precisely, I try to answer the following questions:

1. Do FastSpec and KLEESpectre detect any Spectre vulnerability in the AES implementation?
2. Do FastSpec and KLEESpectre provide the same results?
3. Comparing the results obtained on AES in its OpenSSL v1.0.0[2] implementation (last revised in 2015, before the discovering of Spectre) in its OpenSSL v3.0.0 implementation (the latest version), are there any differences?

The code and the libraries used to answer these questions are available at this GitHub repository: `https://github.com/magdasolitro/msc-thesis`.

4.3.1 Testing with KLEESpectre

Firstly, we tested the libraries with KLEESpectre: all I had to do was to write a brief piece of C code, where we call the functions needed to set up the AES encryption, and then we call a specific function from KLEE to declare what inputs are not known prior execution.

For example, the program shown below was used to test AES in CBC mode of operation (see Appendix B). The most notable variables are:

- `in`, the message to encrypt. Since we are working with AES-128, the size of the string can be at most 16 characters (C uses the ASCII encoding for strings, where each character is consumes 8 bits).

---

[2] `https://github.com/openssl/openssl/tree/91bad2b09eb2ad77da8aca29f80f2d0677e75423`

- `out`, the array where the result of the encryption will be stored.
- `aes_key`, a so-called "struct", namely a complex data structure of the C language containing different fields. In this case, `aes_key` contains two fields, one for the value of the key and one for the number of rounds.
- `ivec`, the initalisation vector, which is used in many modes of operation to guarantee a certain degree of randomness in the encryption process (see Appendix B).

```c
#include <stdio.h>
#include <openssl/aes.h>
#include <openssl/modes.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <openssl/crypto.h>
#include <openssl/aes.h>
#include <klee/klee.h>
#include "aes_locl.h"

int main(){
    const unsigned char in[AES_BLOCK_SIZE] = "Hello, world!";
    unsigned char out[sizeof(in)];
    size_t len = sizeof(in);
    const unsigned char key[] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55,
               0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
    AES_KEY aes_key;
    unsigned char ivec[AES_BLOCK_SIZE];

    klee_make_symbolic(&ivec, sizeof(ivec), "ivec");

    AES_set_encrypt_key(key, 128, &aes_key);

    AES_cbc_encrypt((const unsigned char *) in, out, len,
            (const AES_KEY *) &aes_key, ivec, (const int ) 1);
}
```

In this program, we use a specific KLEE function called `klee_make_symbolic` to declare what input needs to be abstracted. In this case, we must choose `ivec`: this value is random and must be refreshed at every encryption, so it cannot be determined by simply scanning the source code. Plus, the adversary can carry out a Spectre attack by manipulating it (FURTHER ELABORATION). The function takes the following parameters:

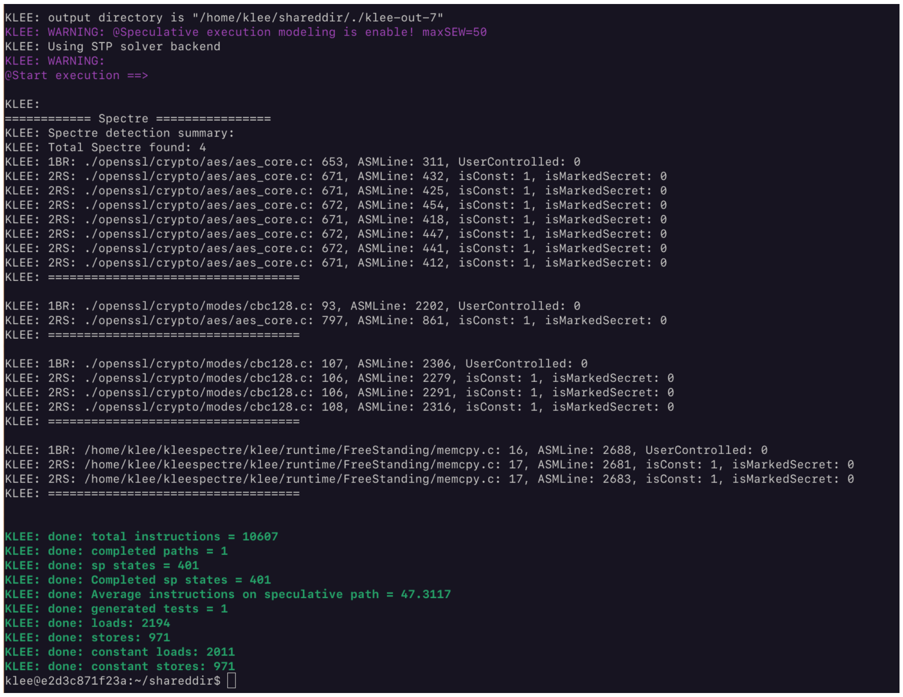
- `&ivec` is the address of the abstract variable;

- `sizeof(ivec)` represents, clearly, its size;
- `"ivec"` is the name with which KLEESpectre will represent the abstract input when printing the output.

After we set the symbolic value, we fill the fields of the `aes_key` struct with OpenSSL function `AES_set_encrypt_key` and perform the encryption with `AES_cbc_encrypt`.

I proceed to illustrate the results obtained for AES in three different modes of operation, namely Cipher Block Chaining (CBC), Output FeedBack (OFB), and Counter (CTR) mode. Instead, I excluded the Electronic CodeBook (ECB) mode because, despite being still in use in certain contexts, it doesn't satisfy the minum security requirements.

Figures 4.4 showS the outcome of KLEESpectre on AES-CBC in OpenSSL v1, finding 4 Spectre vulnerabilities.



```
KLEE: output directory is "/home/klee/shareddir/./klee-out-7"
KLEE: WARNING: @Speculative execution modeling is enable! maxSEW=50
KLEE: Using STP solver backend
KLEE: WARNING:
@Start execution ==>

KLEE:
============ Spectre ================
KLEE: Spectre detection summary:
KLEE: Total Spectre found: 4
KLEE: 1BR: ./openssl/crypto/aes/aes_core.c: 653, ASMLine: 311, UserControlled: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 432, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 425, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 454, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 418, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 447, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 441, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 412, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: ./openssl/crypto/modes/cbc128.c: 93, ASMLine: 2202, UserControlled: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 797, ASMLine: 861, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: ./openssl/crypto/modes/cbc128.c: 107, ASMLine: 2306, UserControlled: 0
KLEE: 2RS: ./openssl/crypto/modes/cbc128.c: 106, ASMLine: 2279, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/modes/cbc128.c: 106, ASMLine: 2291, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/modes/cbc128.c: 108, ASMLine: 2316, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 16, ASMLine: 2688, UserControlled: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 2681, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 2683, isConst: 1, isMarkedSecret: 0
KLEE: ===================================


KLEE: done: total instructions = 10607
KLEE: done: completed paths = 1
KLEE: done: sp states = 401
KLEE: done: Completed sp states = 401
KLEE: done: Average instructions on speculative path = 47.3117
KLEE: done: generated tests = 1
KLEE: done: loads: 2194
KLEE: done: stores: 971
KLEE: done: constant loads: 2011
KLEE: done: constant stores: 971
klee@e2d3c871f23a:~/shareddir$ 
```

Figure 4.4: Results of KLEESpectre on AES-CBC in OpenSSL v1.

As specified in the documentation available at the repository's address, the output should be interpreted as follows:

- the line that starts with `KLEE:1BR` indicates the branch whose misprediction causes cause data leakage;
- the line that starts with `KLEE:2RS` denotes the code line loading the potentially sensitive data;

- the line that starts with `KLEE:3LS` gives the code line information for leaking the sensitive data to cache state.

The first thing I noticed is that there are essentially no differences between the two versions of OpenSSL: both contain apparently 4 Spectre vulnerabilities. The next step is going to the lines indicated in the output and see whether the outcome provided by the tool makes sense or not. The first culprit piece of code is the following:

```
653 if (bits == 128) {
654     while (1) {
655         temp = rk[3];
656         rk[4] = rk[0] ^
657         (Te2[(temp >> 16) & 0xff] & 0xff000000) ^
658         (Te3[(temp >> 8) & 0xff] & 0x00ff0000) ^
659         (Te0[(temp    ) & 0xff] & 0x0000ff00) ^
660         (Te1[(temp >> 24)    ] & 0x000000ff) ^
661         rcon[i];
662         rk[5] = rk[1] ^ rk[4];
663         rk[6] = rk[2] ^ rk[5];
664         rk[7] = rk[3] ^ rk[6];
665         if (++i == 10) {
666             return 0;
667         }
668         rk += 4;
669     }
670 }
671 rk[4] = GETU32(userKey + 16);
672 rk[5] = GETU32(userKey + 20);
```

The leaking branch is in `aes_core.c`, that implements a set of functions used in all modes of operations of AES. More precisely, this branch is a part of the function `AES_set_encrypt_key()`, whose job is to create a struct containing two fields:

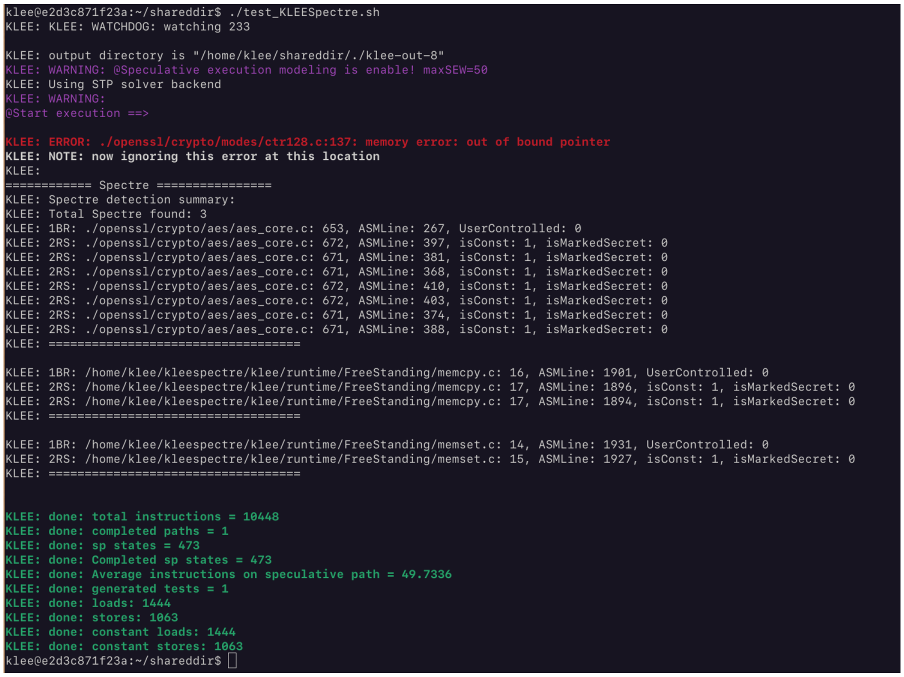
- `rd_key`, representing the value of the key;
- `rounds`, which contains the number of rounds that the AES must perform. This value depends on the number of bits of the key (specified by the parameter `bits`), which can be 128, 192, or 256 (see Appendix B).

The branch is conditioned on the value of `bits`, which represents the length of the key: it's plausible that the leak appears on the branch with guarding expression `bits == 128`, since we are analysing AES 128. The output of KLEESpectre also tells us which line loads potentially sensitive data, which in this case is line 671 and 672. Apparently, this doesn't make much sense: those lines are located *outside* the conditional branch, thus speculative execution does not involve them. However, since KLEESpectre analyses LLVM bitcode, it's appropriate to see which instructions are evaluated as dangerous. Since bitcode files are generated with an encoding that is not human readable, I must run the following command from terminal:

```
llvm-dis testing_AES_CBC.bc -o cbc_readable
```

This command calls the LLVM disassembler, that "takes an LLVM bitcode file and converts it into human-readable LLVM assembly language"[3]. This allows us to ...

I tried to run KLEESpectre also on two more files, `testing_AES_CTR.c` and `testing_AES_OFB.c`, to analyse AES-CTR and AES-OFB respectively: the output is shown in Figures 4.5 and 4.6. However, these inspections don't reveal anything new: the vulnerabilities are always found in `aes_core.c` (which, I remind, defines functions that are used transversely by all AES modes of operations), `memcpy.c` and `memset.c`.



```
klee@e2d3c871f23a:~/shareddir$ ./test_KLEESpectre.sh
KLEE: KLEE: WATCHDOG: watching 233

KLEE: output directory is "/home/klee/shareddir/./klee-out-8"
KLEE: WARNING: @Speculative execution modeling is enable! maxSEW=50
KLEE: Using STP solver backend
KLEE: WARNING:
@Start execution ==>

KLEE: ERROR: ./openssl/crypto/modes/ctr128.c:137: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE:
============ Spectre ================
KLEE: Spectre detection summary:
KLEE: Total Spectre found: 3
KLEE: 1BR: ./openssl/crypto/aes/aes_core.c: 653, ASMLine: 267, UserControlled: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 397, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 381, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 368, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 410, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 403, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 374, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 388, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 16, ASMLine: 1901, UserControlled: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 1896, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 1894, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: /home/klee/kleespectre/klee/runtime/FreeStanding/memset.c: 14, ASMLine: 1931, UserControlled: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memset.c: 15, ASMLine: 1927, isConst: 1, isMarkedSecret: 0
KLEE: ===================================


KLEE: done: total instructions = 10448
KLEE: done: completed paths = 1
KLEE: done: sp states = 473
KLEE: done: Completed sp states = 473
KLEE: done: Average instructions on speculative path = 49.7336
KLEE: done: generated tests = 1
KLEE: done: loads: 1444
KLEE: done: stores: 1063
KLEE: done: constant loads: 1444
KLEE: done: constant stores: 1063
klee@e2d3c871f23a:~/shareddir$
```

Figure 4.5: Results of KLEESpectre on AES-CTR in OpenSSL v1.

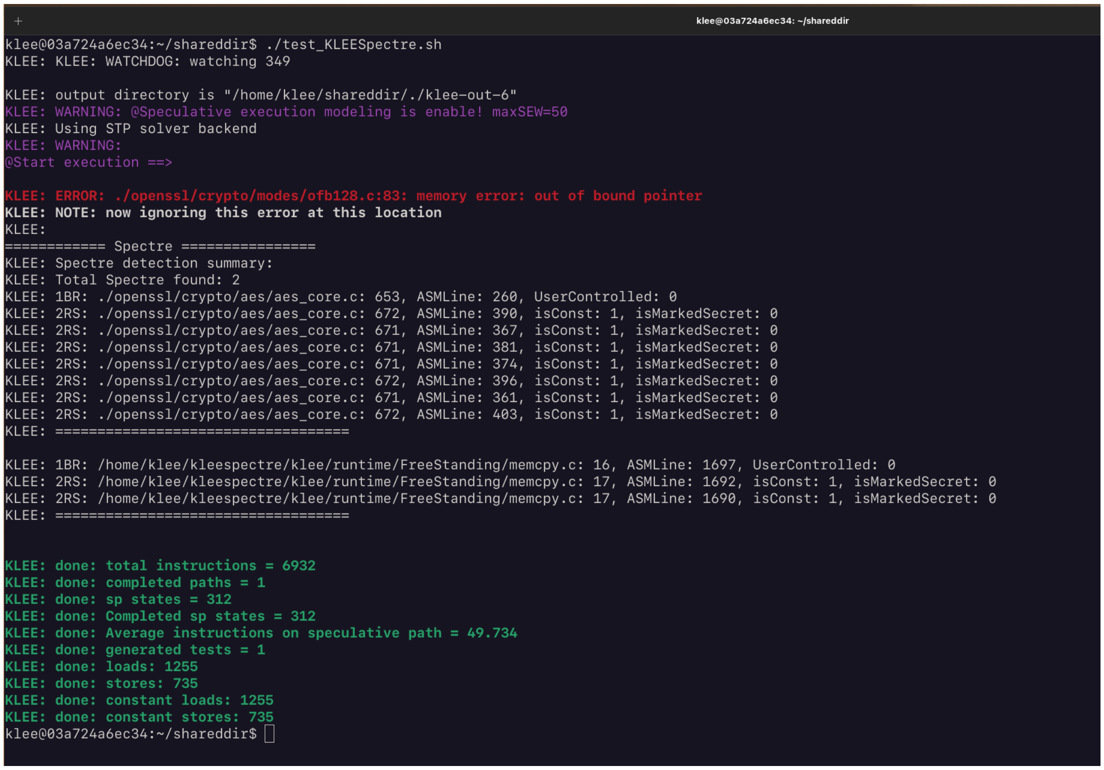
4.3.2 Testing with FastSpec

The second part of this thesis concerns testing AES in different modes of operation with FastSpec.

4.3.3 Consideration about the results

---

[3] https://llvm.org/docs/CommandGuide/llvm-dis.html



```
klee@03a724a6ec34:~/shareddir$ ./test_KLEESpectre.sh
KLEE: KLEE: WATCHDOG: watching 349

KLEE: output directory is "/home/klee/shareddir/./klee-out-6"
KLEE: WARNING: @Speculative execution modeling is enable! maxSEW=50
KLEE: Using STP solver backend
KLEE: WARNING:
@Start execution ==>

KLEE: ERROR: ./openssl/crypto/modes/ofb128.c:83: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE:
============ Spectre ================
KLEE: Spectre detection summary:
KLEE: Total Spectre found: 2
KLEE: 1BR: ./openssl/crypto/aes/aes_core.c: 653, ASMLine: 260, UserControlled: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 390, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 367, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 381, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 374, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 396, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 671, ASMLine: 361, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: ./openssl/crypto/aes/aes_core.c: 672, ASMLine: 403, isConst: 1, isMarkedSecret: 0
KLEE: ===================================

KLEE: 1BR: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 16, ASMLine: 1697, UserControlled: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 1692, isConst: 1, isMarkedSecret: 0
KLEE: 2RS: /home/klee/kleespectre/klee/runtime/FreeStanding/memcpy.c: 17, ASMLine: 1690, isConst: 1, isMarkedSecret: 0
KLEE: ===================================


KLEE: done: total instructions = 6932
KLEE: done: completed paths = 1
KLEE: done: sp states = 312
KLEE: done: Completed sp states = 312
KLEE: done: Average instructions on speculative path = 49.734
KLEE: done: generated tests = 1
KLEE: done: loads: 1255
KLEE: done: stores: 735
KLEE: done: constant loads: 1255
KLEE: done: constant stores: 735
klee@03a724a6ec34:~/shareddir$ 
```

Figure 4.6: Results of KLEESpectre on AES-OFB in OpenSSL v1.

Chapter 5

Conclusions

Appendix A

Deep Learning

A.1 Generative Adversarial Networks (GANs)

The Generative Adversarial Network [13] is a learning framework designed for the task of *generative modelling*, an unsupervised learning task that involves the automatic discovering and learning of patterns in data distributions in such a way that the model can generate new examples that plausibly could have been drawn from the original data set. The innovative idea behind GANs lies in the learning strategy, that sees two actors involved:

- a **Generative** net, G, that generates data;
- a **Discriminator** net, D, that evaluates these data.

Intuitively, the Discriminator should be able to distinguish authentic data from artificially generated data. The goal of the Generator is to produce, starting from a random data distribution, increasingly better data that will eventually fool the Discriminator. From a game theory point of view, the convergence of a GAN is reached when the generator and the discriminator reach a *Nash equilibrium* [35].

Generator To produce a certain kind of data, we assume that there exists a probability distribution (referred to as **target distribution**, p_t) that describes that data we want to create: the points of the distribution represent vectors of features that characterize that type of data. The goal of the Generator is to be able to sample a random noise, z, from a normal or uniform distribution (denoted by p_g) and turn it into a point x following the target distribution, through a function denoted by $Gen()$:

$$x = Gen(z) \text{ with } z \sim p_g$$

Conceptually, z represents the latent features of the images generated, for example, the colour or the shape. In fact, the space from which the Generator samples the seed z is also referred to as **latent space**.

Discriminator The Generator, alone, can only produce some random noise: the purpose of the Discriminator is to guide the Generator to create data that look more similar to the real ones. It's fundamental to underline that before the beginning of the adversarial exchange between the two nets, the Discriminator must be trained (in a supervised fashion) with a data set of real and fake data. For each point in the training set, the net outputs the value $D(x)$, representing the probability that x is real. Thus, what we *ideally* want to obtain at the end of the training is that:

$$D(x) = \begin{cases} 1 & \text{if the input is real} \\ 0 & \text{otherwise} \end{cases}$$

Concretely, during the training, the goal is to *maximize* the ability of discerning real data from fake ones:

$$\max_D V(D) = \mathbb{E}_{x\sim p_t(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[1 - (\log D(G(z)))]$$

where $\mathbb{E}$ denotes the expected value.

To measure the loss, it is possible to use **cross-entropy**, the most popular loss function for learning algorithms that use gradient descent.

Adversarial learning Once the Discriminator is properly trained, the Generator enters into play, and the adversarial confrontation begins. The goal of the Generator is to minimize the difference between the data it generates and real data:

$$\min_G V(G) = \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Overall, the two nets play a so-called **minimax game**:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_t(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

where $V(D,G)$ represents the value function.

The last equation makes it clear why this style of training is called *adversarial*: while the Generator tries to minimise a function, the Discriminator tries to maximise it, thus the two networks have opposite goals. (maybe explain better..?)

A.2 Bidirectional Encoder Representations from Transformers (BERT)

BERT [9] is a deep learning technique for Natural Language Processing (NLP) proposed by Google in 2018. The goal of this framework is to "learn a language" by understanding of how words fit into different contexts. It's interesting to reflect upon how this mechanism resembles the way non-native speakers learn the meaning of particular words in a new language: it is well known that certain expressions don't have a faithful translation in other languages. For this reason, non-native speakers learn what these words stand for simply by observing in what context they are used and what emotional characterisation they are accompanied by. A similar idea applies also to BERT, that uses transformers (described in the next section) to accomplish this task.

A.2.1 Transformers

Transformers [39] were introduced in 2017 by a team at Google Brain to improve certain limitations of recurrent neural networks (RNNs), such as LSTMs. The typical architecture is depicted in Figure A.1.

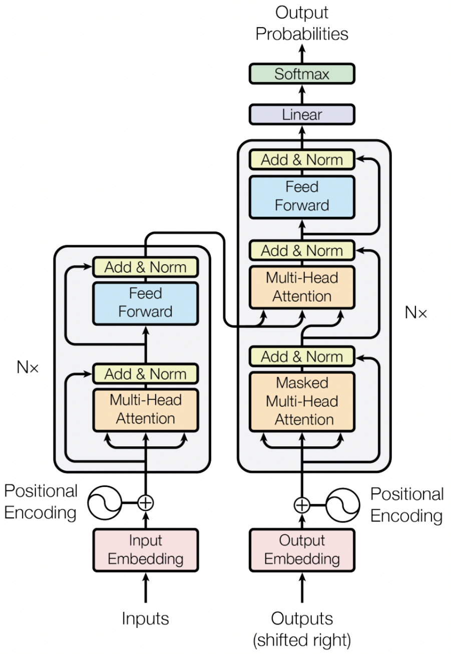

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Figure A.1: Transformers architecture (source: [39]). On the left side there is the encoder, while on the right there is the decoder.

.

Input processing Transformers are typically used for machine translation tasks. For this reason, the input is usually a sentence, namely a set of words, that need to be converted in a mathematical representation in order to be processed by the network. The sentence in a language $\mathcal{L}_1$ is thus passed to the Input Embedding layer, which produces a *vector embedding* for each word. Vector embeddings are a common way to represent words, that tend to assign close vectors to words that have a similar meaning. However, this representation is not enough: as we said in previously in this section, the meaning of a words depends also from the context in which it is placed. For this reason, the vector embeddings are combined with a Positional Encoding, namely a vector that gives context based on the position of the word in a sentence.

After this processing, we obtain a vector representation of the initial sentence that takes into consideration not only the meaning of a words, but also the context in which it appears.

Encoder Once the input processing phase is completed, the vectors are passed to the encoding block, whose goal is learn the characteristics of the source language and the context of a sentence. The encoder consists in two main units:

1. a **Multi-head Attention** layer, that computes an attention vector for each word of the sentence: assuming that the sentence contains n words, the attention vector for the i-th word is an n-dimensional vector, where the j-th coefficient represents how important the j-th words is to determine the meaning of the i-th word;

2. a **Feed-Forward** layer, that is applied to each attention vector, in order to transform them in a way that is more easy to process for the decoder block.

Decoder The decoder is feed with the translation of the sentence that was given in input to the encoder and the same processing steps are performed to transform the sentence into its most accurate mathematical representation. The decoder block has then three main components:

1. a **Masked Multi-head Attention** layer, that masks those words in the destination language that haven't been translated. To understand why this layer is necessary, remember that, when we train a transformer, we have at our disposal (1) a whole sentence in the source language (that is fed into the encoder) and (2) the whole translation of the original sentence in the destination language (that is provided to the decoder). To generate the i-th word in the destination language, the transformer it is allowed to use (1) *all* the words from the input sentence but (2) *only* the $(i-1)$ previous words of the sentence in the destination language! For this reason, all the words from position i onwards are "masked", i.e. are substituted with a special token (usually `<MASK>`).

2. a **Multi-head Attention** layer, which takes in input (1) the embedding vectors produced by the encoder and (2) the attention vectors produced by the previous layer of the decoder. The output is a vector where each coefficient expresses how the corresponding word relates to the other elements of the sentence, both in the source and in the destination language.

3. a **Feed-Forward** layer, that has the same purpose as the one in the encoder block.

The output of the decoder is a vector of probabilities: the i-th coefficient corresponds to the probability that the i-th words is the next word in the translated sentence.

A.2.2 Architecture

As we said in the previous section, the two block of a transformer accomplish different functions:

- the **encoder** learns what is the source language and what is context;
- the **decoder** learns how to map words in the source language into words of the destination language.

BERT was designed to solve problems (such as sentiment analysis, question answering, text summarisation, etc.) that have to do with understanding language and context, while the translation part is irrelevant. Therefore, BERT's architecture is just made by a stack of encoders.

A.2.3 Training

The training phase consists in two parts: *pre-training* and *fine-tuning*. The first part is needed to make BERT learn what is language and what is context. Once this phase is completed, we must to fine-tune our model to accomplish a specific task, for example sentiment analysis.

Pre-training Firstly, BERT is trained on two unsupervised task simultaneously: Masked Language Model (MLM) and Next Sentence Prediction (NSP). Note that these tasks are not BERT's final goal! These tasks are only needed to make BERT understand the language, but the training on the objective task will be made only during the fine-tuning phase. During the MLM task, we provide BERT with sentences where random words have been masked and the network must learn to guess the correct word in each place. In order to succeed, BERT is forced to learn the notion of context *within the same sentence* and to evaluate whether a word can fit into a given context or not. For the NSP task, BERT receives in input two sentences with a temporal connection, meaning that one of the two sentences only makes sense when placed before or after the other (for example, when reading $A =$ "This is John" and $B =$ "He is my friend", we can immediately see that B makes sense only if it's preceeded by A). BERT must learn to figure out what sentence comes after the other one: in this way it comprehends context even *across different sentences*.

Fine-tuning Once BERT has been sufficiently trained, we can fine-tune it to accomplish a more specific task, for instance:

- classification tasks such as **sentiment analysis**, which is done similarly to the NSP task.
- **question answering**...
- ...

Appendix B

Block ciphers and AES

The goal of this appendix is to provide the basic notions concerning block ciphers and modes of operations that are fundamental to understand how AES works. Given the breadth of the topic, here we will discuss only the concepts that are strictly necessary to understand the core of this thesis, but the curious readers can rely on the references given in the rest of the appendix to dig deeper into the topic.

The appendix begins by presenting some game-based security notions (maybe not), necessary to understand how the security of ciphers is evaluated. Then we briefly describe block ciphers in Section B.1, before proceeding with Section B.2, where we present how a block cipher can be used repeatedly on different part of the message to produce the final encryption. Finally, we have all the tools to reach Section B.3, concerning the history and the design of AES.

B.1 Block ciphers

A block cipher is a deterministic algorithm F that operates on data units of fixed length, called *blocks*. F accepts in input a data block of size n, and performs a transformation using a key k, producing in output a block of same size as the input. More formally, F is defined as follows:

$$F : \{0,1\}^{|k|} \times \{0,1\}^n \to \{0,1\}^n$$

where $|k|$ represents the key length and n the size of the block. The definition given above means that a ciphertext c is obtained from a key k and a message m (also called "plaintext") as:

$$F(k,m) = c$$

Since the key remains fixed for a certain number of encryptions, before being refreshed, we can parametrise F on k, defining $F_k(m) \overset{\text{def}}{=} F(k,m)$.

If F represents the encryption algorithm, then the decryption algorithm is obtained simply by computing F^{-1}, thus:

$$F^{-1}(k,c) = m$$

B.2 Modes of operation

There is a clear limitation of block ciphers: they can only encrypt data of fixed size. While the case in which $|m| < n$ can be easily circumvented through padding, if $|m| > n$ we would be forced to discard a part of the message. *Modes of operation* address this problem, providing a way to securely and efficiently encrypt long messages using block ciphers. In simple terms, a mode of operation defines how to concatenate the use of a block cipher in such a way that each cipher encrypts a specific part of the message. In general, if the message has size $|m| > n$, then m is split in $\ell = \frac{|m|}{n}$ different blocks, m_1, m_2, ... , m_ℓ, and the whole ciphertext is obtained as:

$$c = \langle F(k, m_1), F(k, m_2), ..., F(k, m_\ell) \rangle$$

where the operator $\langle \cdot, ..., \cdot \rangle$ represents a a generic combination. For the sake of brevity, the rest of this section will present only the modes of operation that were tested in this thesis. Those who are interested in other modes can find them in the references provided throughout this Appendix.

B.2.1 Cipher Block Chaining (CBC) mode

To encrypt using this mode, we choose uniformly at random an initialization vector (IV) of length n, which is unique for every encryption. Then, we generate the ciphertext blocks by applying the block cipher to the XOR of the current plaintext block and the previous ciphertext block. Figure B.1 visually depicts the process. Mathematically, the ciphertext is the result of the following operations:

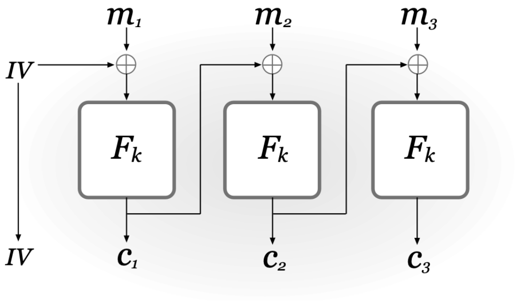


Figure B.1: Visual representation of Cipher Block Chaining (CBC) mode (source: [16])

$$\begin{cases} c_1 = F_k(IV \oplus m_1) \\ c_i = F_k(c_{i-1} \oplus m_i) \text{ for all } i > 1 \end{cases}$$

Using a random initialisation vector introduces an element of randomness in the encryption process and makes it not deterministic: the repeated encryption of the same plaintext results in different ciphertext. We will not dive into the discussion about the security of modes of operations, as it is beyond the scope of this appendix. However, it's sufficient to mention that introducing an element of randomness during the encryption ensures to a greater level of security with respect to deterministic schemes. Clearly, the choice of the IV plays a crucial role: it should be chosen uniformly at random and it must be included in the ciphertext to allow a correct decryption.

Despite being considered secure, this mode of operation presents a mild disadvantage: since the encryption of block i depends on the value of the previous ciphertext, the computation of the final ciphertext can't be parallelised, thus it's not the most efficient choice.

B.2.2 Output Feedback (OFB) mode

The Output FeedBack (OFB) mode works similarly to the CBC mode, performing some operations in a different order. Figure B.2 is a high-level depiction of how the OFB mode works. This mode of operation, like Cipher Block Chaining, involves a random initialisation vector, whose value must be unpredictable to guarantee the minimum security requirements.

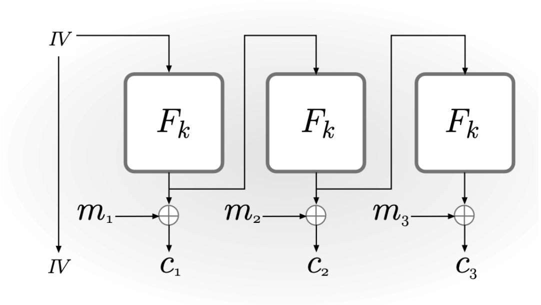


Figure B.2: Visual representation of Output Feedback (OFB) mode mode (source: [16])

Mathematically, the ciphertext is computed as follows:

$$\begin{cases} c_1 = F_k(IV) \oplus m_1 \\ c_i = F_k(c_{i-1}) \oplus m_i \text{ for all } i > 1 \end{cases}$$

This mode of operation presents the same disadvantage of the CBC-mode, because the computation can't be parallelised.

B.2.3 Counter (CTR) mode

In the CTR mode of operation, a counter $ctr \in \{0,1\}^n$ is sampled uniformely at random. Then, the encryption unfolds as follows:

$$c_i = F_k(ctr + i) \oplus m_1$$

where *ctr* and i are viewed as integers and addition is done modulo 2^n. Figure B.3 depicts In contrast to all the modes discussed previously, the CTR mode has the advantage that encryption and decryption can be fully parallelised, since all the blocks can be computed independently of each other.

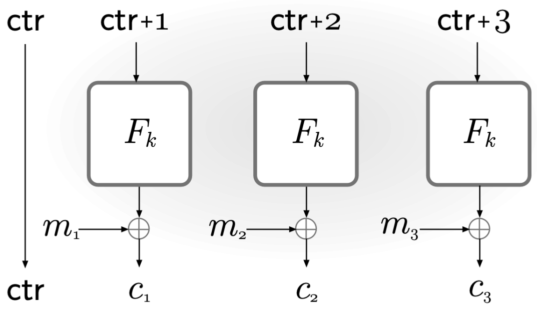


Figure B.3: Visual representation of Counter (CTR) mode mode (source: [16])

B.3 AES: Advanced Encryption Standard

AES [16] [1] is a block cipher that was born in 1998 as replacement of the obsolete DES (Data Encryption Standard) [27], on request of the U.S. government. At first, its use was limited to military purposes, but nowadays it's one of the most common block ciphers in use, deeply embedded in various applications.

The AES cipher has a 128-bit block length and can use 128-, 192-, or 256-bit keys; depending on the key length, the algorithm performes 10, 12 or 14 rounds respectively. During the execution, AES works on a 4×4 array of bytes, called *state*, and each step of the algorithm performs a certain trasformation on this state. More precisely, the main operations present in the algorithm are:

1. KeyExpansion: Derives fresh 128-bit key at each round, using the AES key schedule.
2. AddRoundKey: Combine each byte of the 4×4 state matrix with a byte of the selected key using the XOR ($\oplus$) operation.
3. SubBytes: Performs a non-linear substitution where each byte is replaced with another according to a lookup table.
4. ShiftRows: Performs a transposition where the last three rows of the state are shifted cyclically of a certain number of steps.
5. MixColumns: Performs a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.

Each operation is executed multiple times, according to the order represented in Algorithm 1. In this algorithm, we denote with $nRounds$ the number of rounds executed according to the key length, with K the key derived from the AES key schedule at each round, with S the state during each step of the execution, with $SBox$ the non-linear transformation performed during SubBytes, with $nRows$ the amount of shift in the ShiftRow operation, and with m the invertible linear transformation used in MixColumns.

Algorithm 1 Advanced Encryption Standard (AES)

```
for $i = 0$ to $nRounds - 1$ do
    $K \leftarrow$ KeyExpansion()
    $S \leftarrow$ AddRoundKey($K$)
    $S \leftarrow$ SubBytes($Sbox$)
    $S \leftarrow$ ShiftRows($nRows$)
    $S \leftarrow$ MixColumns($m$)
end for
$S \leftarrow$ SubBytes($Sbox$)
$S \leftarrow$ ShiftRows($nRows$)
$S \leftarrow$ MixColumns($m$)
```

Bibliography

[1] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[2] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. 51*, 3 (2018).

[3] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (2008), pp. 387–401.

[4] Bayer, U., Comparetti, P., Hlauschek, C., Krügel, C., and Kirda, E. Scalable, behavior-based malware clustering.

[5] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, 2008), OSDI'08, USENIX Association, p. 209–224.

[6] Canella, C., Bulck, J. V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 249–266.

[7] Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 2 (1936), 345–363.

[8] Dai, P., Pan, Z., and Li, Y. A review of researching on dynamic taint analysis technique. In *Proceedings of the 2018 3rd Joint International Information Technology Mechanical and Electronic Engineering Conference (JIMEC 2018)* (2018/12), Atlantis Press, pp. 118–123.

[9] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[10] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. *SIGPLAN Not. 53*, 2 (mar 2018), 693–707.

[11] Fog, A. The microarchitecture of intel, amd, and via cpus. an optimization guide for assembly programmers and compiler makers, May 2021.

[12] Gomes, I. V., Morgado, P., Gomes, T., and Moreira, R. M. L. M. An overview on the static code analysis approach in software development.

[13] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. *Advances in neural information processing systems 27* (2014).

[14] Hinman, P. G. Fundamentals of mathematical logic. *Bulletin of Symbolic Logic 13*, 3 (2007), 363–365.

[15] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, aug 2019), USENIX Association, pp. 621–637.

[16] Katz, J., and Lindell, Y. *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman Hall/CRC, 2014.

[17] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (jul 1976), 385–394.

[18] Kocher, P. Spectre mitigations in microsoft's c/c++ compiler, 2018.

[19] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 1–19.

[20] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies* (USA, 2018), WOOT'18, USENIX Association, p. 3.

[21] Krinke, J. Information flow control and taint analysis with dependence graphs.

[22] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[23] LO, O., BUCHANAN, W. J., AND CARSON, D. Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa). *Journal of Cyber Security Technology 1*, 2 (2017), 88–107.

[24] MAISURADZE, G., AND ROSSOW, C. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 2109–2122.

[25] MANGARD, S. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *Information Security and Cryptology — ICISC 2002* (Berlin, Heidelberg, 2003), P. J. Lee and C. H. Lim, Eds., Springer Berlin Heidelberg, pp. 343–358.

[26] MANGARD, S., OSWALD, E., AND POPP, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[27] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Data encryption standard (des). FIPS Publication 46-3, October 1999.

[28] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on aes. In *Selected Areas in Cryptography* (Berlin, Heidelberg, 2007), E. Biham and A. M. Youssef, Eds., Springer Berlin Heidelberg, pp. 147–162.

[29] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

[30] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *Security and Privacy in the Age of Ubiquitous Computing* (Boston, MA, 2005), R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, Eds., Springer US, pp. 295–307.

[31] ODIFREDDI, P. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers.* Sole Distributors for the Usa and Canada, Elsevier Science Pub. Co., 1989.

[32] ORS, B., GURKAYNAK, F., OSWALD, E., AND PRENEEL, B. Power-analysis attack on an asic aes implementation. vol. 2, pp. 546 – 552 Vol.2.

[33] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive 2002* (01 2002), 169.

[34] PERLEBERG, C. H., AND SMITH, A. J. Branch target buffer design and optimization. Tech. Rep. UCB/CSD-89-552, EECS Department, University of California, Berkeley, Dec 1989.

[35] Ratliff, L. J., Burden, S. A., and Sastry, S. S. Characterization and computation of local nash equilibria in continuous games. In *2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (2013), pp. 917–924.

[36] Rogers, H. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, MA, USA, 1987.

[37] Tol, M. C., Gulmezoglu, B., Yurtseven, K., and Sunar, B. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings.

[38] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[39] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

[40] Wang, G., Chattopadhyay, S., Biswas, A. K., Mitra, T., and Roychoudhury, A. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Trans. Softw. Eng. Methodol. 29*, 3 (jun 2020).

[41] Wang, G., Chattopadhyay, S., Gotovchits, I., Mitra, T., and Roychoudhury, A. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering PP* (11 2019), 1–1.

[42] Xavier Rival, K. Y. *Introduction to Static Analysis: an abstract interpretation perspective.* The MIT Press, Massachusetts, 2020.

[43] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 719–732.

[44] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, Association for Computing Machinery, p. 116–127.

[45] Zhang, Y. Cache side channels: State of the art and research opportunities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2617–2619.