

# Detecting Microarchitectural Attacks

DRAFT

Author: *Magdalena M. Solitro*



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Static analysis for cybersecurity</b>	<b>5</b>
2.1	Theoretical foundations . . . . .	6
2.2	Practical approaches and related challenges . . . . .	8
2.2.1	Taint analysis . . . . .	8
2.2.2	Symbolic execution . . . . .	10
<b>3</b>	<b>Spectre</b>	<b>13</b>
3.1	Background . . . . .	13
3.1.1	Out-of-Order execution and micro-operations . . . . .	14
3.1.2	Speculative execution . . . . .	14
3.1.3	Cache side channels . . . . .	15
3.2	Spectre variants . . . . .	16
3.2.1	Spectre-PHT (Pattern History Table) . . . . .	17
3.2.2	Spectre-BTB (Branch Target Buffer) . . . . .	18
3.2.3	Spectre-RSB (Return Stack Buffer) . . . . .	19
3.2.4	Spectre-STL (Store To Load) . . . . .	20
<b>4</b>	<b>Verification tools to detect Spectre</b>	<b>23</b>
4.1	FastSpec . . . . .	24
4.2	KLEESpectre . . . . .	25
4.3	Testing AES . . . . .	27
<b>A</b>	<b>Complexity theory</b>	<b>29</b>
A.1	Complexity classes . . . . .	29
A.2	Boolean satisfiability problem: SAT . . . . .	29
<b>B</b>	<b>Deep Learning</b>	<b>31</b>
B.1	Generative Adversarial Networks (GANs) . . . . .	31
B.2	Natural Language Processing (NLP) . . . . .	32

<b>C</b>	<b>Block ciphers and AES</b>	<b>33</b>
C.1	Block ciphers . . . . .	33
C.2	Modes of operation . . . . .	33
C.2.1	Electronic Code Book (ECB) mode . . . . .	34
C.2.2	Cipher Block Chaining (CBC) mode . . . . .	35
C.2.3	Output Feedback (OFB) mode . . . . .	35
C.2.4	Counter (CTR) mode . . . . .	36
C.3	AES: Advanced Encryption Standard . . . . .	36

# Notation

This chapter constitutes a reference for all the notational conventions and the mathematical symbols used throughout this work, especially in the Static Analysis chapter.

$\mathcal{L}$             a generic programming language

$\mathcal{P}$             a generic property, i.e. generic a set of programs



# Chapter 1

## Introduction





# 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 [40].

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,*

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

*The halting problem is not computable: an algorithm such as `halt` does not exist, as proved simultaneously by Alonso Church [7] and Alan Turing [37] 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,*

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

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

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

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 feasible 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 existence of a research field dedicated to static analysis.

What we just said implies that the results provided by the algorithm will occasionally produce unreliable 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 concern 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 [11]. 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 desirable 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:

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

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:

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

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 comes with 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 the details, but the interested readers can find out more about the topic in [29], [34], [13].

## 2.2 Practical approaches and related challenges

This section discusses the general ideas behind some of the most effective 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.

### 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. 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 during the program execution 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 [27], malware analysis [4] [42], and web applications [3] [28], 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 [27]:

```

1  if(x==0){
2      y=0;
3  } else if (x==1){
4      y=1;
5  } else if (x==2){
6      y=2;
7  } ...

```

This conditional structure goes on in the form

```

1  if(x == n){
2      y = n;
3  }

```

This program is semantically equivalent to the following instruction:

```

1  y = x;

```

However, a taint analysis will mark *y* as tainted in the last instruction, but not in the first example. Under-tainting affects TaintCheck [27], even though the authors proposed some mitigations to the problem.

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  $u \rightarrow v$  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 [39] use a conservative approximation of the CFG, which can contain more edges than 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: (TO BE REVIEWED)

**Definition 4** (Control Dependency Graph (CDG)). A *control dependency graph (CDG)* [19] is a directed graph  $G = (V, E)$  in which nodes represent the program's statements or expressions, while an edge  $x \rightarrow y$  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] [15]: 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, parameters 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*  $\sigma$ , that maps the variables of the program either with expressions over concrete values or with symbolic values.
- Some *path constraints*  $\pi$ , 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:

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

As you can see, `a` is a parameter 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  $\alpha_a$ : 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).  $\pi$  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:

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

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  $\alpha_a \neq 0$ :

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

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

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 encountered 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), 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* [17] and *Meltdown* [20]. These two attacks were independently discovered by various researchers and were published in two works, [17] and [20], destined to leave a permanent mark in the field of cybersecurity. More precisely, [17] 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 [20] 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<sup>1</sup>.

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. [17], "Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels."

---

<sup>1</sup>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  $\mu$ -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 [10]. 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  $\mu$ -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  $\mu$ -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  $\mu$ -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 important 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 [43] 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 [31], trace-driven [31], 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:

```

1  square-and-multiply(M, e, N){
2      R = 1
3      for (i=0 to n-1){
4          R = R^2 mod N
5          if(e[i] == 1){ R = R * M mod N }
6      }
7      return R
8  }
```

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 symmetric encryption, to exfiltrate entire bytes of the symmetric key [21] [23] [24] [30]. 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" [26]. 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 [41], 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 [41], 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.

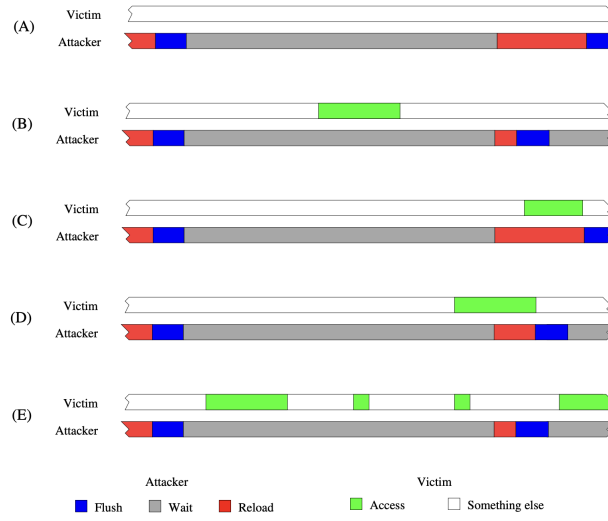


Figure 3.1: FLUSH+RELOAD time measurement [41]. (A) and (B) represent the main cases described above. (C), (D), (E) are more peculiar cases.

### 3.2.1 Spectre-PHT (Pattern History Table)

Spectre-PHT [17] [6] [9], 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 misprediction.

**Pattern History Table (PHT)** The Pattern History Table [10] 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]:

```
1 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 perform 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:

```

1  if (x < len(array)) {
2      array[x] = value;
3  }

```

Just as the described above, the attacker can "train" the CPU to guess a certain outcome of the branching instruction and then exploits the misprediction 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 [17] [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 [32] 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" [17].

### 3.2.3 Spectre-RSB (Return Stack Buffer)

Spectre-RSB [22] [18] [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 [18]. One way to carry out the attack is to exploit the stack overflow 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 one 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 the *data flow*. More precisely, this version takes advantage of the memory disambiguation mechanism that is put in place by most modern processors.

**Memory disambiguation** Memory disambiguation<sup>2</sup> 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 techniques, 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:

```
1  store $1, 2($2)    # Mem[R2+2] <= R1
2  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

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Memory\\_disambiguation](https://en.wikipedia.org/wiki/Memory_disambiguation)



resolving ambiguous dependencies require more sophisticated techniques than static dependency, and this is where the memory disambiguation mechanism comes into play.

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 were 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 [36], a recent tool that uses deep learning techniques not only to detect Spectre, but also to generate new gadgets using a GAN-based framework named SpectreGAN. On the other hand we have KLEESpectre [38], 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<sup>1</sup>, a popular software library used to ensure secure communication over the network. More precisely, we tackle the following questions:

1. Do FastSpec and KLEESpectre detect any Spectre vulnerability? If so, what variant do they detect?
2. Do FastSpec and KLEESpectre provide the same results?
3. Comparing the results obtained on hash functions implemented in OpenSSL v1.0.0 (last revised in 2015, before the discovering of Spectre) and those implemented in OpenSSL v3.0.0 (the latest version), are there any differences?

As specified in [36], FastSpec was already used to test OpenSSL v3.0.0 on public-key algorithms and procedures for digital signatures, namely RSA, EDDSA and DSA. Our work targets one specific symmetric-key cipher, AES (Advanced Encryption Standard, see Appendix C), in different modes of operation.

The rest of the chapter is organised as follows: Section 4.1 and 4.2 provide an overview on SpectreGAN and KLEESpectre, 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.

---

<sup>1</sup>Official website: <https://www.openssl.org/>

## 4.1 FastSpec

SpectreGAN is a Generative Adversarial Network that was presented along with FastSpec [36], a tool based on deep learning that is able to identify Spectre gadgets much faster than common rule-based methods. The need to employ deep learning techniques comes from a significant limitation encountered in other analysis tools: the detection of vulnerable code sections often relies on a pattern matching process that compares these sections with known gadgets, which amount to only 17 (15 by Kocher [16] and 2 modified example introduced by Spectector). Despite being conceptually sound, this strategy falls short at recognising subtle code variations occurring in different programs. Furthermore, such a low number of gadgets makes it difficult to perform a comprehensive analysis of the tools.

The authors of FastSpec target this problem through the following steps:

- They extend the current set of 15 gadgets to 1 million by applying a technique called *mutational fuzzing*, creating an appropriate dataset for the deep learning algorithm;
- They describe a specific implementation of a GAN, named *SpectreGAN*, that is trained on the dataset mentioned in the previous point to learn the distribution of existing Spectre gadgets;
- They introduce *FastSpec*, a tool based on supervised neural network embeddings, to identify potential gadgets faster than rule-based methods.

The rest of the section is focused on the first two points, namely in the parts that deal with the generation of new gadgets, as FastSpec is out of our scope of interest. All the background notions about the deep learning techniques used within this tool are discussed in Appendix B.

The authors propose two ways of producing new gadgets starting from the initial set of 17: the first one leverages on random insertion of instructions, and is used to create a large dataset of code snippets. The second one uses the previously generated dataset to train a Generative Adversarial Network: the goal is to learn the leakage distribution of Spectre vulnerabilities, to be able to "intelligently" produce vulnerable code.

**Mutational fuzzing** Firstly, the initial set of gadgets is increased by compiling each code snippet with three different compilers and different optimisation options, obtaining 6 assembly functions for each gadget. Then, the authors describe an algorithm that performs the following steps on each gadget:

1. It inserts random assembly instructions in random locations of the gadget,
2. It checks whether the functional properties of the gadgets are preserved,
3. It checks whether the new gadget still leaks secrets through speculative execution.

Step 2) is particularly important because it verifies whether the newly inserted instructions improperly modify the flags that are checked in the conditional jumps: if that is the case, the secret may be leaked without even exploiting speculative execution.

**SpectreGAN** As the authors clarify in their work, "*the purpose of SpectreGAN is to develop an intelligent way of creating assembly functions instead of randomly inserting instructions and operands*" [36]. In this way, they don't simply modify the gadgets with arbitrary changes, but they build a system that learns the features that characterise a vulnerable piece of code and is able to reproduce them in a brand new code snippet.

Both the Generator and the Discriminator are implemented with seq2seq, an architecture commonly used for machine translation tasks (see Appendix B). As for the **generation phase**, a gadget is sampled from the dataset, divided in tokens, and "masked", i.e. some arbitrary tokens in the gadget are hidden. The Generator learns the leaking patterns by trying to fill correctly the masked positions. As for the **discrimination phase**, the Discriminator is fed with 1) the embedding vectors corresponding to the original tokens, and 2) the embedding vectors corresponding to the *masked* tokens ...

## 4.2 KLEESpectre

KLEESpectre [39] 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.

**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 in Figure ... A standard symbolic execution engine like KLEE would explore the paths shown in Figure 4.1 : 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 }

```

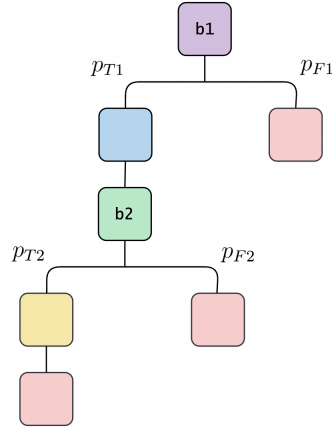


Figure 4.1: Standard symbolic execution of the program.

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

3.  $sp_{T1}$ :  $x \geq \text{SIZE}$  is satisfiable and the branch **b1** is **mispredicted**. In this case, KLEESpectre forks a new state with constraint  $x \geq \text{SIZE}$ , but proceeds by executing the code fragment highlighted in **blue**.
4.  $sp_{F1}$ :  $x < \text{SIZE}$  is satisfiable and the branch **b1** is **mispredicted**. KLEESpectre forks a new state with constraint  $x < \text{SIZE}$ , but proceeds by executing the code fragment highlighted in **pink**.

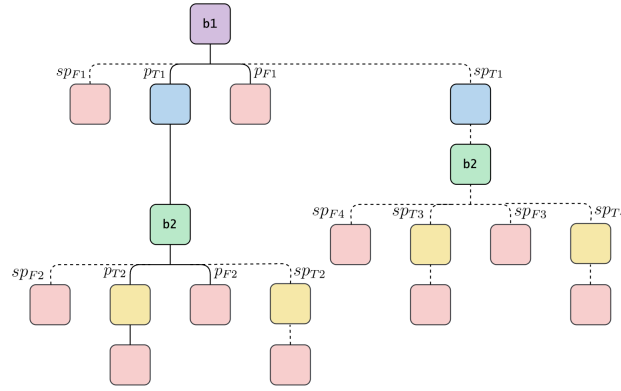


Figure 4.2: Speculative symbolic execution of the program.

The decision tree produced by KLEESpectre as depicted in Figure 4.2

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 is also able to handle 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 4.2 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, see Section 2.2.1).

**Cache modelling** Speculative symbolic execution is further extended with cache modelling: in simple terms, this means that KLEESpectre models the cache behaviour of an execution path (concrete or speculative) to verify whether a secret value remains in the cache when the program terminates.

To do this, KLEESpectre distinguishes between two different types of 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.

## 4.3 Testing AES





# Appendix A

## Complexity theory

### A.1 Complexity classes

### A.2 Boolean satisfiability problem: SAT



# Appendix B

## Deep Learning

### B.1 Generative Adversarial Networks (GANs)

The Generative Adversarial Network [12] 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* [33].

**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..?)

## B.2 Natural Language Processing (NLP)

# Appendix C

## Block ciphers and AES

This appendix begins with a brief introduction to block ciphers and modes of operations, before diving deeper into the details of AES.

### C.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:

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

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 plaintext  $p$  as:

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

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

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

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

### C.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  $|p| < n$  can be easily circumvented through padding, if  $|p| > n$  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  $p$  has size  $m > n$ , the  $P$  is split in  $\ell = \frac{m}{n}$  blocks,  $p_1, p_2, \dots, p_\ell$ , and the whole ciphertext is obtained as:

$$c = \langle F(k, p_1), F(k, p_2), \dots, F(k, p_\ell) \rangle$$

where the operator  $\langle \cdot, \dots, \cdot \rangle$  represents a generic combination. In the rest of the section, we will describe some of the most popular modes of operation.

### C.2.1 Electronic Code Book (ECB) mode

This is the simplest mode of operation: it consists in performing the encryption of the data blocks separately, and then concatenating the results, as shown in Figure C.1. Therefore, the whole ciphertext is computed as

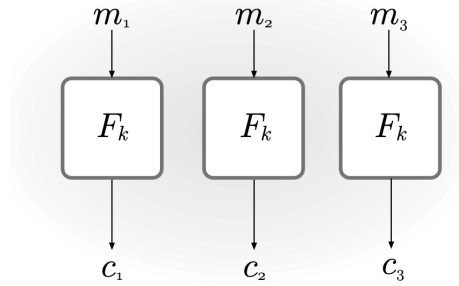


Figure C.1: Visual representation of Electronic Code Book (ECB) mode (source: [14])  
fv

$$C = F(k, p_1) \parallel F(k, p_2) \parallel \dots \parallel F(k, p_\ell)$$

where the operator  $\parallel$  represents a concatenation.

This mode of operation is not used in real-world applications: since it's deterministic (i.e. by encrypting the same plaintext multiple times, we always obtain the same ciphertext), it can be proved that it doesn't satisfy the minimum standards of security. For instance, let's consider the *indistinguishability* property of ciphertexts, which is informally defined as follows:

**Definition 5** (Indistinguishability). Two encryptions are said to be *indistinguishable* if it's infeasible for the adversary to distinguish pairs of ciphertexts based on the message they encrypt.

Since the ECB mode of operation is deterministic, this property is trivially broken, making ECB intrinsically insecure. Providing a formal argument of why it is considered "insecure" is out of the scope of our work, but the interested readers can refer to [14] (Section 3.6.2 "Block-Cipher Modes of Operation"), [25], [35] for a detailed discussion on the topic.

### C.2.2 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 C.2 visually depicts the process. Mathematically, the

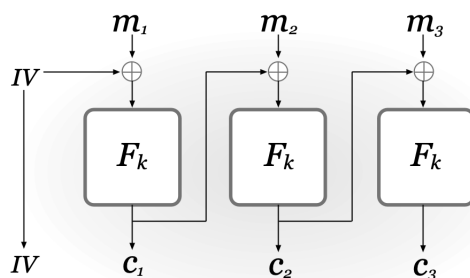


Figure C.2: Visual representation of Cipher Block Chaining (CBC) mode (source: [14])

ciphertext is the result of the following operations:

$$\begin{cases} c_1 = F_k(IV \oplus p_1) \\ c_i = F_k(c_{i-1} \oplus p_i) \text{ for all } i > 1 \end{cases}$$

This mode of operation is not deterministic, as the initialisation vector introduces an element of randomness in the encryption. As opposed to the ECB mode, this means that the repeated encryption of the same plaintext results in different ciphertext. This characteristic allows the CBC mode to satisfy the basic standards of security (more precisely, it is CPA-secure<sup>1</sup>). Clearly, the choice of the  $IV$  plays a fundamental role in the encryption: in order to ensure the indistinguishability of ciphertexts, it's fundamental to choose it uniformly at random.

### C.2.3 Output Feedback (OFB) mode

The Output FeedBack (OFB) mode works similarly to the CBC mode, performing some operations in a different order. Figure C.3 is a high-level depiction of the OFB mode. This mode of operation, like Cipher Block Chaining, also involves a random initialisation vector, whose value must be unpredictable to guarantee the minimum security requirements.

Mathematically, the ciphertext is computed as follows

$$\begin{cases} c_1 = F_k(IV) \oplus p_1 \\ c_i = F_k(c_{i-1}) \oplus p_i \text{ for all } i > 1 \end{cases}$$

<sup>1</sup>[https://en.wikipedia.org/wiki/Chosen-plaintext\\_attack](https://en.wikipedia.org/wiki/Chosen-plaintext_attack)

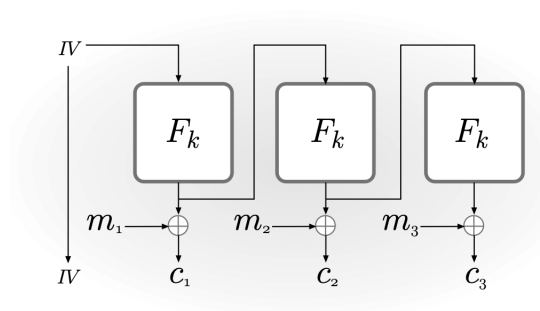


Figure C.3: Visual representation of Output Feedback (OFB) mode (source: [14])

#### C.2.4 Counter (CTR) mode

### C.3 AES: Advanced Encryption Standard

AES [14] [1] is a block cipher that was born in 1998 as replacement of the obsolete DES, 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. In particular, AES is a subset of the Rijndael ciphers, a family of ciphers with different key and block sizes.

The AES cipher has a 128-bit block length and can use 128-, 192-, or 256-bit keys. In contrast with DES, which uses a Feistel structure, AES is essentially a so-called Substitution-Permutation network. AES's algorithm consist of five main steps:

1. **KeyExpansion:** a fresh 128-bit key is derived at each round, using the AES key schedule.
2. **AddRoundKey:** in every round of AES, a 128-bit sub-key is derived from the master key, and is interpreted as a 4-by-4 array of bytes. The state array is updated by XORing it with this sub-key.
3. **SubBytes:** ...
4. **ShiftRows:** ...
5. **MixColumns:** ...

Added to In the final round, MixColumns is replaced by AddRoundKey. Usually, the number of rounds goes from 10 to 14.



# 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., JOVANOVIĆ, 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] 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.

- [10] FOG, A. The microarchitecture of intel, amd, and via cpus. an optimization guide for assembly programmers and compiler makers, May 2021.
- [11] GOMES, I. V., MORGADO, P., GOMES, T., AND MOREIRA, R. M. L. M. An overview on the static code analysis approach in software development.
- [12] 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).
- [13] HINMAN, P. G. Fundamentals of mathematical logic. *Bulletin of Symbolic Logic* 13, 3 (2007), 363–365.
- [14] KATZ, J., AND LINDELL, Y. *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman Hall/CRC, 2014.
- [15] KING, J. C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (jul 1976), 385–394.
- [16] KOCHER, P. Spectre mitigations in microsoft’s c/c++ compiler, 2018.
- [17] 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.
- [18] 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.
- [19] KRINKE, J. Information flow control and taint analysis with dependence graphs.
- [20] 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).
- [21] 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.
- [22] 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.

- [23] 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.
- [24] MANGARD, S., OSWALD, E., AND POPP, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, 1st ed. Springer Publishing Company, Incorporated, 2010.
- [25] MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 2018.
- [26] 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.
- [27] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.
- [28] 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.
- [29] 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.
- [30] ORS, B., GURKAYNAK, F., OSWALD, E., AND PRENEEL, B. Power-analysis attack on an asic aes implementation. vol. 2, pp. 546 – 552 Vol.2.
- [31] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive 2002* (01 2002), 169.
- [32] 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.
- [33] 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.
- [34] ROGERS, H. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

- [35] SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 2015.
- [36] TOL, M. C., GULMEZOGLU, B., YURTSEVEN, K., AND SUNAR, B. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings.
- [37] TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265.
- [38] 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).
- [39] 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.
- [40] XAVIER RIVAL, K. Y. *Introduction to Static Analysis: an abstract interpretation perspective*. The MIT Press, Massachusetts, 2020.
- [41] 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.
- [42] 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.
- [43] 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.