SpecTacle - A platform agnostic analysis tool for detecting Spectre-PHT gadgets in binaries

Pedro Miguel Sousa Bernardo

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Pedro Miguel dos Santos Alves Madeira Adão
Prof. Daniel Gruss

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. Marco Guarnieri

November 2021

To Carolina and my family

For their role in allowing me to achieve my dreams.

Acknowledgments

I would like to thank Martin Schwarzl for his invaluable insights, tips, and always being available. I would also like to thank my supervisors Pedro Adão, who was consistently available throughout the development of this thesis and who provided great advice and counseling, and Daniel Gruss for giving me the opportunity and the honor of working with his team at the Institute of Applied Information Processing and Communications of the Graz University of Technology.

A special thank you to my wonderful girlfriend Carolina, for her patience and unconditional support. I also want to thank my family for supporting me throughout my studies and always pushing me to do my best. Finally, a big thank you to my friends who became my second family during my studies abroad.

Resumo

Os ataques *Spectre* exploram uma das técnicas mais eficazes de otimização dos processadores (CPU): a execução especulativa. Instruções executadas especulativamente podem deixar vestígios nas estruturas da microarquitetura. Estes vestígios podem ser posteriormente recuperados, usando *side-channels*, permitindo a atacantes ler memória confidencial. Como mitigação a ataques *Spectre*, os fabricantes de CPUs propõem uso de instruções de serialização que inativam a execução especulativa para um determinado segmento de código. No entanto, para aplicar esta técnica oferecendo simultaneamente um equilíbrio entre segurança e performance, excertos de código vulnerável, denominados de *Spectre gadgets*, têm de ser localizados.

Nesta tese, contribuímos para o estado-da-arte da deteção de *Spectre gadgets*, implementando a ferramenta *SpecTacle*. Esta ferramenta extrai caminhos de execução potencialmente vulneráveis aos quais aplica *taint analysis* estática, o que permite a deteção de gadgets produzindo um baixo número de falsos negativos, sem sacrificar os tempos de execução. Adicionalmente, estes ”potenciais” gadgets são validados através de execução simbólica. Por fim, avaliamos a eficácia do *SpecTacle* através de *litmus tests* e aplicamos a ferramenta, com sucesso, a um caso real.

Adaptámos também a nossa ferramenta para suportar a deteção de gadgets que exfiltram informação através de um *side-channel* baseado na unidade AVX2, usados em ataques *NetSpectre*, algo que não é suportado por nenhuma outra ferramenta. Para finalizar, propomos ainda uma abordagem que permite detetar, de forma estática, *Spectre* gadgets em código compilado JIT, algo que até agora era apenas feito de forma dinâmica, e validamos esta abordagem através de uma prova de conceito num programa Java.

Palavras-chave: execução especulativa, *side-channel*, análise de programas, gadgets Spectre

Abstract

Spectre attacks exploit one of the most effective CPU optimization techniques: speculative execution. Instructions executed speculatively can leave traces in microarchitectural structures that attackers can later recover using side-channels, enabling attackers to read arbitrary data. The adoption of speculative execution is widespread across today's CPU architectures, which means billions of devices are susceptible to Spectre attacks. As a result, CPU manufacturers have suggested using serializing instructions that effectively disable speculative execution for a given code segment as mitigation. However, to apply this technique while providing a good balance between security and performance, these vulnerable code snippets, called Spectre gadgets, must be first located.

In this thesis, we contribute to the state-of-the-art in Spectre gadget detection. We implement SpecTacle, a tool that extracts potentially vulnerable control flow paths and performs static taint analysis to detect gadgets in such paths while producing a low number of false negatives without sacrificing execution time. Additionally, these potential gadgets are validated through symbolic execution. Finally, we show SpecTacle's efficacy through litmus tests and successfully apply it to a real-world example.

In addition to the detection of Spectre-PHT gadgets, we adapt our tool to support the detection of gadgets that leak data through an AVX-based side-channel, not supported by any other state-of-the-art tool. These gadgets are used by NetSpectre, a Spectre variant that enables remote Spectre attacks. Finally, we propose an approach to statically detecting Spectre gadgets in JIT-compiled code, which until now has only been done dynamically, and validate this approach through a proof-of-concept.

Keywords: speculative execution, side-channel attacks, program analysis, Spectre gadgets

Contents

Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xv
List of Figures . . . xvii
Nomenclature . . . 1
Glossary . . . 1

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Thesis Outline . . . 3

2 Background **5**
2.1 Out-of-Order Execution . . . 5
2.2 Branch Prediction . . . 6
2.2.1 Dynamic Branch Prediction . . . 6
2.3 Speculative Execution . . . 8
2.4 Cache . . . 8
2.4.1 Cache Mappings . . . 8
2.4.2 Hierarchy . . . 9
2.4.3 Replacement Policies . . . 9
2.4.4 Cache Attacks . . . 9
2.5 Spectre Attacks . . . 11
2.5.1 Spectre Variants . . . 11
2.5.2 Spectre PHT . . . 13
2.5.3 NetSpectre . . . 14
2.5.4 Spectre Countermeasures . . . 16
2.6 Meltdown Attacks . . . 17
2.7 Program Analysis . . . 17
2.7.1 Taint Analysis . . . 18

2.7.2 Symbolic Execution . . . 19
2.7.3 Fuzzing . . . 20
2.7.4 Intermediate Representations . . . 20
2.7.5 Just-in-time compilation . . . 20

3 Related Work **23**
3.1 SpecFuzz . . . 23
3.2 SpecTaint . . . 24
3.3 FastSpec . . . 25
3.4 oo7 . . . 26
3.5 Pitchfork . . . 26
3.6 KLEESpectre . . . 27
3.7 SPECTECTOR . . . 27
3.8 Binsec/Haunted . . . 28
3.9 Speculator . . . 28
3.10 Dynamic Process Isolation . . . 29

4 Implementation **31**
4.1 High-level Overview . . . 31
4.1.1 Pre-processing and Locating Conditional Branches . . . 31
4.1.2 Explorer . . . 33
4.1.3 Executor . . . 33
4.1.4 Validator . . . 33
4.2 Pre-processing . . . 33
4.2.1 Java Code Analysis . . . 34
4.2.2 FastSpec Integration . . . 35
4.2.3 Locating Conditional Branches . . . 37
4.3 Explorer . . . 37
4.4 Executor . . . 38
4.4.1 Analysis Hooks . . . 41
4.4.2 Taint Policy . . . 42
4.5 Gadget Detection Analyses . . . 45
4.5.1 DefaultCacheEncodingAnalysis . . . 46
4.5.2 ComparisonResultAnalysis . . . 49
4.5.3 NetSpectreAnalysis . . . 51
4.6 Validator . . . 52
4.7 Optimizations . . . 54
4.7.1 Memory Load Limit . . . 54
4.7.2 Parallelization . . . 54
4.7.3 State Memoization . . . 54

5 Results 55
5.1 Problem Description 55
5.2 Technologies 56
5.3 Experimental Setup 57
5.4 Performance and Scalability 57
5.4.1 Explorer 57
5.4.2 Validator 59
5.5 Gadget Identification 60
5.5.1 Dataset 60
5.5.2 Evaluation 61
5.6 Case Studies and Extensions 62
5.6.1 Java Example 63
5.6.2 AVX-based NetSpectre Example 64
5.6.3 Brotli 65

6 Conclusions 67
6.1 Future Work 68

Bibliography 69

List of Tables

5.1 SpecTacle results for a speculation window of 15 instructions using the DefaultTaintPolicy, without running the Validator. 58

5.2 SpecTacle results for a speculation window of 35 instructions using the DefaultTaintPolicy, without running the Validator. 58

5.3 SpecTacle results for a speculation window of 35 instructions using the StrictTaintPolicy, without running the Validator. 58

5.4 SpecTacle results for a speculation windows of 15 and 35 instructions using the DefaultTaintPolicy and running the Validator. Entries marked * represent values that we could not obtain. 59

5.5 Table comparing the execution time and number of functions with flagged gadgets for FastSpec with a confidence level of 0.48, and SpecTacle using the StrictTaintPolicy for a speculation window of 35 instructions, with and without the Validator. Entries marked * represent values that we could not obtain. 61

5.6 Execution times of FastSpec[19] and SpecTacle for selected Phoronix Test Suite binaries. SpecTacle was run with a speculation window size of 35 instructions and the DefaultTaintPolicy (DTP). * means that the specific value could not be obtained 62

5.7 Comparison between FastSpec's precision and recall values for confidence levels of 0.48, 0.6 and 0.8, with SpecTacle's using the DefaultTaintPolicy for a speculation window of 35 instructions and without the Validator. **P** and **R** represent precision and recall respectively. 63

5.8 Comparison of the precision and recall values, alongside execution time of the DefaultTaintPolicy and the StrictTaintPolicy without the Validator, and the DefaultTaintPolicy with the Validator (V) in the last column. Results obtained with a speculation window of 35 instructions. **P** and **R** represent precision and recall, respectively. * means that it was not possible to obtain a result. 63

List of Figures

2.1 Memory access times of cache hits and cache misses for the Intel i5 2540M CPU 10
2.2 Example of encoding data into the cache using Spectre-PHT. 14
2.3 NetSpectre leak gadget. Leaks 1 bit from a bitstream based a user-provided value `x` [25]. 15
2.4 NetSpectre leak gadget using an AVX-based covert-channel to encode confidential data [25]. 16

3.1 Side-by-side of native code and code produced by the SpecFuzz custom compiler [16] . . 24
3.2 FastSpec training gadget [68]. 26

4.1 High level schematic of SpecTacle's execution flow 32
4.2 FastSpec's Execution flow 36
4.3 FastSpec's execution flow modified to retrieve the function labels and generate a *gadgets* file. Labels marked with * represent modified script files. 36
4.4 Spectre-PHT gadget CFG using basic blocks recovered by Capstone [84] 38
4.5 Spectre-PHT gadget CFG using IRSBs recovered using angr [82] and PyVEX [85] 39
4.6 Side-by-side of the v01 gadget from Kocher [41], introduced in the Spectre [3] paper, and the v10 gadget variant from Kocher [41] 46
4.7 Example of a possible v10 gadget's control-flow graph 50
4.8 High-level ComparisonResultAnalysis analysis state machine. **1)** In state 1 ComparisonResultAnalysis is waiting for a conditional branch. **2)** While in state 2, ComparisonResultAnalysis is looking for a tainted guard. **3)** While in state 3, if a LDle statement is visited, a potential v10 gadget is flagged. 51
4.9 NetSpectre leak gadget that uses the cache as a covert channel [46]. 51
4.10 Possible execution paths following a conditional branch C. 52
4.11 Scheme of how the Validator simulates speculative execution and validates the feasibility of a given transient path. 53

5.1 Spectre-PHT gadget injected from Kocher [41] in the Phoronix Test Suite mbw benchmark [103]. 61
5.2 NetSpectre gadgets [46] added to the litmus test and used for SpecTacle's NetSpectre gadget detection proof-of-concept. 65
5.3 Spectre-PHT gadget found in Brotli by SpecTaint [92] [23] 66

Chapter 1

Introduction

1.1 Motivation

Gordon Earle Moore observed in 1965 that the number of transistors in integrated circuits doubles every two years and predicted this would continue in the future. This effect was labeled *Moore's Law* and held throughout the past decades. However, recent developments have been challenging this prediction, as stated by Carl Anderson in 2009 [1]. Central Processing Unit (CPU) clock frequencies and transistor counts are stabilizing due to physical constraints like overheating. Increasing performance-per-watt values has also become a priority in recent years partly due to climate and energy consumption issues, paired with the continuous quest for performance increases. These factors contributed to the development of alternate ways of increasing CPU performance by manufacturers, which come in the form of both architectural and microarchitectural optimizations.

The introduction of data and instruction *caches* is responsible for significant performance increases in CPUs by bridging the gap between memory access times and CPU performance [2]. Optimizations such as *out-of-order* execution allow for *instruction-level parallelism* by enabling different stages of the process of executing an instruction to be performed in parallel by scheduling these stages in a *pipeline* [2]. This scheduling increases the number of instructions executed in a CPU clock cycle, directly increasing CPU performance. However, in some situations, the CPU cannot perform this scheduling until a specific instruction stage is resolved, which causes the pipeline to stall temporarily. One of these situations, called a *control hazard*, can occur when conditional branching instructions are executed. Conditional branches lead execution to different code locations, depending on a given condition. The calculation of this condition can involve memory accesses, which can be impossible to resolve when the CPU adds a new instruction to the pipeline. Speculative execution enables the CPU to guess where execution will continue based on a history of previous branching instructions kept by microarchitectural structures. In the worst case, the CPU makes an incorrect guess, which is called a *branch misprediction*. After a branch misprediction is recognized, the CPU discards the speculatively executed instructions, restores the CPU state altered by these instructions, and resumes execution in the correct path. However, not all state is restored. Cache entries, for example, remain unaltered. Instructions that modify microarchitectural state

that is not subsequently restored are called *transient instructions*.

Today's CPU architectures widely adopt speculative execution. These architectures include those developed by the leading CPU manufacturers, such as AMD, ARM, and Intel. However, in 2018 transient execution attacks were introduced: Spectre, Meltdown, Foreshadow, and Foreshadow NG [3–6]. These attacks exploit transient execution to break data isolation boundaries and leak sensitive information. Since transient instructions are a consequence of speculative execution, and speculative execution is employed in the majority of today's devices, this means that billions of devices are vulnerable to transient execution attacks [7].

Spectre [3] attacks use specific instruction sequences called *Spectre gadgets* to exploit branch prediction mechanisms causing the execution of transient instructions that leak arbitrary data through side-channels, most commonly cache-based side-channels. This data can then be retrieved by attackers through cache timing attacks like Flush+Reload [8] or Prime+Probe [9]. Several Spectre variants have been introduced. Spectre-PHT (or Spectre V1) exploits the *Pattern History Table* to cause branch mispredictions. Other variants exploit different structures like the *Branch Target Buffer* (Spectre-BTB or V2 [3]), the *Return Stack Buffer* (Spectre-RSB or V3 [3, 10]), or memory disambiguation techniques like *Store-To-Load Forwarding* (Spectre-STL or V4 [3, 11]).

Mitigations for Spectre have been proposed, both at hardware and software levels. Disabling speculative execution seems to be the easiest and most effective mitigation, but this comes at the cost of significant performance decreases. CPU manufacturers such as Intel have also proposed mitigations consisting of micro-code updates that make it harder for attackers to influence internal CPU structures or that reduce the attack range of Spectre attacks [12] by restricting the memory that can be accessed during speculative execution. Ultimately, these countermeasures do not eliminate speculative execution attacks. InvisiSpec [13] and SafeSpec [14] propose the addition of internal CPU structures that are specific to speculative execution and that are invisible to attackers, which would mitigate all speculative execution attacks. However, these solutions are expensive, partly due to the extra hardware requirement, and have not yet been applied in the real world. As a result, hardware-level Spectre mitigations are still being developed.

Mitigations at the software level, like the insertion of serializing instructions, like memory load fences (`lfence`), cause performance losses but are more straightforward to implement than hardware mitigations. However, to correctly apply these mitigations, we must first find the Spectre gadgets present in programs since adding these instructions near every branching instruction would induce significant performance penalties and is practically equivalent to disabling speculative execution. For these reasons, Spectre gadget detection is currently a very prominent research topic.

Many approaches have already been explored recently [15–20], each with its advantages and disadvantages. These approaches range from dynamic analysis techniques like *fuzzing* or the use of *hardware performance counters*, to static analysis techniques like pattern matching and *taint tracking*. Most research argues that a combination of static and dynamic analysis is necessary to achieve the desired balance between efficacy and performance. However, a complete end-to-end approach combining these types of analysis has not yet been developed.

Research in Spectre gadget detection has produced tools that either monitor a program's execution during runtime (Speculator [21], Dynamic Process Isolation [22]), analyze native executable files (SpecFuzz [16], SpecTaint [23], oo7 [20], Binsec/Haunted [18], etc.), or analyze assembly code [19]. However, Spectre gadget detection in interpreted or JIT-compiled code has not been explored enough. Additionally, even though cache-based Spectre gadgets are the most common, other possible side-channels, like port contention [24] or delays in the AVX unit [25] can be exploited in Spectre gadgets. Research in the detection of Spectre gadgets that encode confidential data through these alternative side-channels is also lacking.

1.2 Contributions

Our main goal with this work is to make progress towards an end-to-end solution for automatic Spectre gadget detection. For that purpose, we propose SpecTacle, a static analysis tool for detecting Spectre-PHT gadgets. SpecTacle has three main modules:

- An *Explorer* module that recovers execution paths that may contain Spectre gadgets.
- An *Executor* module that simulates those paths statically and propagates taint information according to a given taint policy, and that supports arbitrary analyses that reason about the Executor's state, including taint values.
- *Validator* module based on symbolic execution to discard false positives flagged by the previous static analysis, increasing precision.

Since our work is focused mainly on the detection of Spectre-PHT gadgets, we concentrate our efforts on the *Executor* module and in developing an efficient static analysis module that produces the lowest amount of false negatives without sacrificing precision and computation resources.

We adapt SpecTacle to be able to detect NetSpectre leak gadgets that use an AVX-based side-channel [25], which, to the best of our knowledge, is not supported by any other state-of-the-art tool.

Finally, we propose a static approach to detecting Spectre gadgets in JIT-compiled code, which has only been done dynamically to the best of our knowledge. Additionally, we prove the efficacy of this approach through a proof-of-concept that successfully detects a Spectre-PHT gadget in a Java program.

1.3 Thesis Outline

In Chapter 2 of this thesis, we provide an overview of the theoretical background behind this work. This background includes the necessary concepts of computer architecture, transient execution attacks, and program analysis techniques. Chapter 3 discusses related work, i.e., other state-of-the-art tools, their approach to Spectre gadget detection, and their corresponding pros and cons. Chapter 4 explores the implementation of SpecTacle in detail, its architecture and analysis techniques. Chapter 5 discusses the

experimental results and how SpecTacle fairs against other state-of-the-art tools. Finally, in Chapter 6 we discuss our findings and future work that can help improve our results and advance the state-of-the-art.

Chapter 2

Background

In this chapter, we discuss the background required for this thesis.

In Section 2.1 we introduce the concept of out-of-order execution. Section 2.2 explains the basics of branch predictors and why they help greatly increase CPU performance. Section 2.2 provides an overview of caches and cache attacks that allow attackers to leak cache data through timing attacks. In Section 2.3 we briefly introduce the concept of speculative execution. Sections 2.5 and 2.6 provide overviews of Spectre [3], Meltdown [4] and Microarchitectural Data Sampling attacks, their variations, and how they exploit speculative execution. Section 2.7 provides an overview of the main program analysis techniques used in state of the art.

2.1 Out-of-Order Execution

Modern CPUs execute instructions in stages in order to implement *instruction-level parallelism*. These stages belong to a pipeline whose purpose is to allow for different stages of various instructions to be executed simultaneously, saving overall execution time [26]. Examples of these stages are: *fetch*, *decode*, *execute* and, *memory access*.

Instruction pipelining assumes that each instruction completes before the next one begins, which is not always accurate. Situations where this assumption does not hold, are called hazards [26].

There can be **structural**, **control** and **data** hazards [26].

- **Structural hazards** occur when multiple instructions in the pipeline require the same resource.
- **Control hazards** (or branch hazards) occur when the control flow is modified. If the guard of a conditional branch is not resolved, the CPU does not know which instructions to execute next, so a control hazard occurs.

- **Data hazards** occur when an instruction in the pipeline depends on the result of a previous instruction. There are three types of data dependencies that can cause data hazards: *read-after-write*, *write-after-write* and *write-after-read*.

These hazards, in particular control hazards, can lead to CPU stalls which significantly reduce performance. So that other instructions do not have to wait for hazards to be resolved, independent instructions, i.e., instructions that do not lead to data or structural hazards, can be executed out-of-order to maximize the instruction throughput of the pipeline. Instructions must be fetched and decoded in order, to ensure that a program's behavior is not affected by out-of-order execution. This order is kept by using the `reorder buffer` (ROB), by saving both the intermediate results and operands of the instructions that are executed out-of-order [26]. If an exception occurs, it is trivial to refer to the reorder buffer to ignore all instructions executed after the instruction that caused the exception. Otherwise, all execution results are committed in order [26].

Although instructions can be discarded, the microarchitectural state, i.e., the cache, is not reverted. Instructions that are not committed but that leave microarchitectural side-effects are called *transient instructions* [4].

2.2 Branch Prediction

Control hazards can have a significant negative impact on performance by causing the pipeline to stall [26]. To ensure maximum pipeline throughput, the CPU must fetch an instruction immediately after the previous one was fetched [26]. Fetching the next instruction is a problem for branch instructions because the CPU does not know what path execution will lead. When it comes to conditional branches, these can follow two paths: taken or not taken, while indirect branches can have multiple targets that may only be resolved at runtime. Branch predictors help mitigate this problem.
A branch predictor is a CPU unit that predicts the next instruction for a specific branch given the current program counter. [2]. Branch prediction can be:

- *Static*, when it is performed at compile-time [2]. An example of static branch prediction is a branch predictor that always predicts that a branch is taken. Loop unrolling [2] is also considered a static branch prediction technique.

- *Dynamic*, when the prediction is performed at runtime.

2.2.1 Dynamic Branch Prediction

Dynamic branch predictors make use of internal buffers that are updated at runtime whenever a branch is resolved. These structures include the Branch History Table (BHT) [2].

Branch History Table

The BHT is an internal buffer, indexed by the lower bits of the address of the branch instruction, and it contains information about the recent history for that branch, i.e., if the branch was taken or not taken in the last executions. This information can be comprised of one or more bits.

On a one-bit BHT, the prediction changes whenever there is a misprediction. For example, the BHT contains the value of 0 for a given branch, meaning that it will be predicted `not taken`. If there is a misprediction in this branch, the BHT entry will be changed to 1 [2].

Two-bit branch predictors implement a more complex state machine. The two bits can be interpreted as a counter that increments when a branch is taken and decrements when not taken. This counter can go up to the value of three (or 11) or down to 0. Values of 0 and 1 (00 and 01 in binary) cause a `not taken` prediction, while values of 2 and 3 (10 and 11 in binary) cause a `taken` prediction [2].

Branch Target Buffer (BTB)

Another internal structure of dynamic branch predictors to note is the BTB. The BTB is a cache that holds branch destinations for a given branch [26]. A BTB can hold information such as a tag to identify the given branch, branch target address, and even instruction bytes from the branch target address [27]. The BTB is the first structure queried by the branch predictor. Upon an instruction is fetched from memory, the BTB is queried. If that query results in a hit, then the information in the BTB is used to predict the branch target [27].

Pattern History Table (PTH)

The PTH is used in two-level predictors, as the second level to the BHT, and maps each possible value for a branch in the BHT to another 2-bit counter [28, 29]. The value of the BHT for a given branch is used to index the PHT counter update. The PHT counters implement the same state machine as the BHT counters. [28, 29]

Return Stack Buffer (RSB)

Return instructions are a specific indirect branch type, so there is significant performance to be gained by predicting them. For this purpose, the RSB is used. The RSB holds the return addresses of the most recent function calls. Thus, each time a function call is performed, the expected return address, i.e., the address of the instruction after the call, is added to the RSB and mapped to the given call for later prediction [30].

2.3 Speculative Execution

Speculative execution consists of identifying branch instructions after fetching them, predicting their branch targets in order to avoid or reduce the performance overhead caused by a possible control hazard, and then executing that path speculatively.

The branch predictor is used to predict the branch target. Execution then continues from that address, executing the following instructions out-of-order until the branch is resolved. Intermediate values and instruction order are kept in the Reorder Buffer. When the branch instruction is executed, the results of instructions executed out-of-order are either committed to the corresponding registers or memory or are discarded.

Only a certain number of instructions can be executed while in speculation, and the size of the Reorder Buffer determines this number. To the number of instructions executed in speculation, we call `speculation window` [3].

2.4 Cache

CPU caches are small but fast memory buffers between the CPU and the main memory (DRAM). It exploits the principle of *locality of reference* [26] which consists of the fact that, for a given time frame, CPUs tend to access memory in the same locations. When the CPU accesses a memory address, that address is brought to the cache to speed up memory access times. Subsequent accesses to that address will use the cached version, saving on memory access time.

2.4.1 Cache Mappings

The CPU needs to be able to know if a specific memory address is cached or not. There are several ways in which the main memory (DRAM) can be mapped to the cache. Direct-mapped caches map each DRAM address to a specific cache address. This mapping is usually simple and is often done as follows: `A % N`, where % is the modulo operator, `A` is the DRAM address, and `B` is the number of cache blocks [26]. This calculation can be performed efficiently if the number of cache blocks is a power of two [26]. Direct mapping can lead to collisions because the cache is smaller than the DRAM by design. The CPU must be able to distinguish between memory addresses that are mapped to the same cache line. To allow this for distinction, the cache contains `tags`. Tags only need to contain the upper bits of an address that are not used to index the cache [26].

There is another mapping scheme where addresses can be mapped to any block in the cache. This scheme is called *fully associative*. For locating specific blocks in the cache, since these can be placed anywhere in the cache, the entire cache must be searched [26]. For this search to be efficient, it needs to be done in parallel, and for that, special hardware must be used, which increases hardware cost [26].

To combine the best of these two approaches, *set associative* caches were developed. These caches can map each address to a limited number of cache locations. An $\hat{n}$-way set associative cache contains

n positions where each address can be mapped [26]. These n positions belong to a set. Each DRAM address is directly mapped to a set.

2.4.2 Hierarchy

CPUs can have *multilevel caches*, which consists of a memory hierarchy of cache levels [26]. Modern-day CPUs have three cache levels (L1, L2, and L3) [31]. The L1 cache is often smaller but faster. Both L1 and L2 have an instruction and a data cache for caching both data and code separately [31].

Typically, the L1 and L2 caches are distributed per-core, while the L3 cache is shared among all cores [31, 32].

2.4.3 Replacement Policies

When inserting data in the cache, collisions might occur. Replacement policies were developed to resolve cache collisions. A few notable examples of cache replacement policies are *least-recently used* (LRU) and *random*. If all possible cache blocks where an addressed can be mapped to are full, the *random* replacement policy chooses a random block to be replaced. While enforcing the LRU policy, the possible cache location with the highest `age` field, i.e., the cache location last accessed at the longest time, is chosen [26].

The goal of replacement policies is to keep the most needed data on the cache to avoid *cache misses*. A cache miss happens when the CPU tries to access a memory location that is not in the cache. Alternatively, a *cache hit* occurs when the data is found in the cache.

2.4.4 Cache Attacks

The purpose of caches is to reduce memory access times. Cache hits are significantly faster than cache misses. This difference can reach 200 CPU clock cycles as illustrated by fig. 2.1. Naturally, this decrease in memory access times significantly increases CPU performance, but it also introduces a prominent side-channel used in many microarchitectural attacks.

Flush+Reload and Evict+Reload

Flush+Reload [8] was first introduced by Yarom and Falkner [8]. It uses the `clflush` instruction, which evicts a memory line from all cache levels [8]. This eviction serves as the setup for the side-channel. Flush+Reload assumes that both victim and attacker can access the same memory addresses. The attacker flushes a specific memory address, so it is not present in any cache level. If the victim accesses that address, it will be cached. The attacker can then perform a memory address on this same address and measure the access time, for example, with the `rdtsc` instruction [33]. Timing information like the one in fig. 2.1 can be used by an attacker to determine if the access resulted in a cache hit or miss, letting the attacker know if the victim accessed the given memory address (in case of a cache hit) or not (in case of a cache miss).

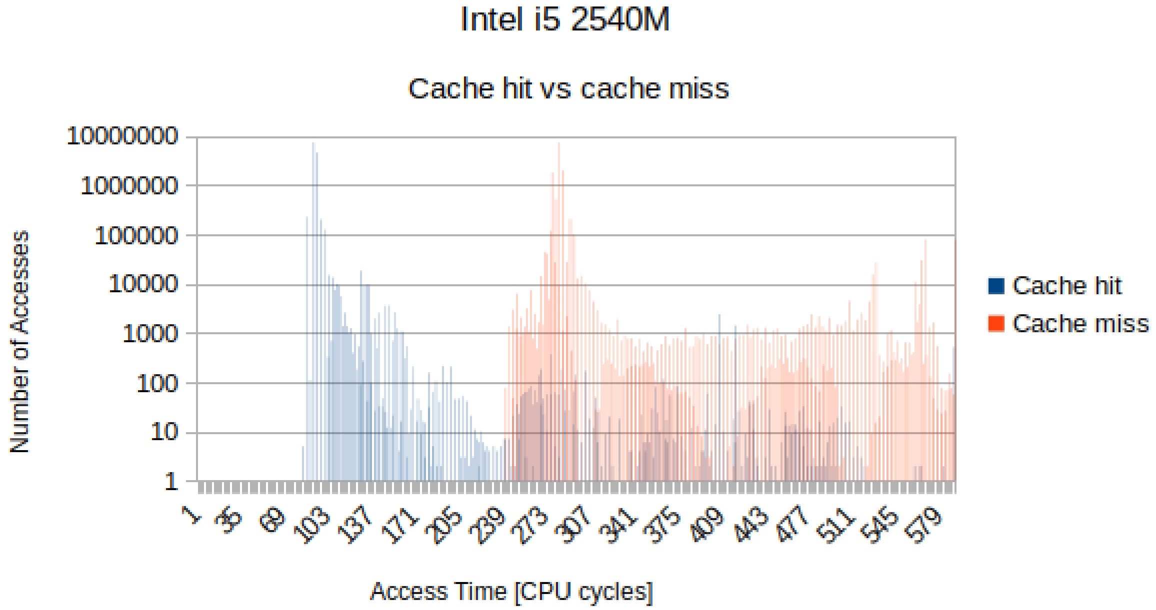

Figure 2.1: Memory access times of cache hits and cache misses for the Intel i5 2540M CPU

Evict+Reload [34] was introduced by Gruss et al. [34] as an alternative to Flush+Reload [8] for situations where the `clflush` instruction is not available and there are no specific architectural interfaces for evicting cache blocks. These situations include Instruction Set Architectures (ISA) that do not support `clflush` such as the case of ARMv7 [35], or remote attacks such as NetSpectre [25] where the attacker does not have access to the victim's system directly. Evict+Reload is also used to mount attacks from JavaScript where a `clflush` instruction is not provided [3].

Evict+Reload [34, 36] follows the same formula as Flush+Reload [8], but since there is no `clflush` instruction, cache block eviction is performed by accessing memory addresses that are congruent to the cache block being evicted for a given replacement policy. Eventually, after enough memory accesses are performed, the cache block is evicted.

Prime+Probe

Prime+Probe [9, 37, 38], generalized by Osvik et al. [9] has the advantage of not requiring the measurement of the victim's execution time. An attacker first populates a cache set. This stage is called *prime*. If the victim accesses data congruent to the data primed by the attacker during its execution, attacker data will be evicted from that cache set to be replaced by the victim's data. Afterward, the attacker can then access that cache, measuring access time. This stage is called `probe`. According to the access times, the attacker can conclude if the victim accessed that cache set.

Flush+Flush

Flush+Flush [39] was first introduced by Gruss et al. [39]. Flush+Flush [39] exploits the timing difference between flushing a cached and an uncached memory address. This timing difference can be exploited in the same way Flush+Reload is, replacing the memory access for a `clflush` instruction.

2.5 Spectre Attacks

In their work from 2018, Kocher et al. [3] introduced a class of attacks called **Spectre**. Spectre attacks exploit transient execution by tricking branch predictors into mispredicting branch targets that are then executed speculatively.

Spectre attacks exploit the fact that, although the architectural state is restored after the CPU resolves a control hazard, the microarchitectural state is not. One particular element of the microarchitectural state that is not reverted is the cache. This work mainly focuses on Spectre attack variants that use the cache as a microarchitectural covert-channel to retrieve leaked information. Cache attacks can be used to extract information from the cache. Examples of these attacks are Flush+Reload [8], Evict+Reload [34, 36] and Flush+Flush [39].

A Spectre attack has two main components: attacker-controlled branch mispredictions and a sequence of instructions that encode data in a covert channel.

Transient instructions can violate program semantics because they execute in a context that is not architecturally possible. Due to this, they can, for example, access data that would be out of bounds in a standard execution flow. Thus, if an attacker can combine instructions that encode data in a cover-channel with instructions that perform out-of-bounds memory accesses by tricking the branch predictor into executing these instructions in the wrong context, he can mount Spectre attack.

2.5.1 Spectre Variants

Variant 1: Spectre-PHT

Spectre-PHT attacks were first introduced by Kocher et al. [3]. These attacks cause branch mispredictions by poisoning the Pattern History Table, which branch predictors refer to alongside the Branch History Buffer (BHB) to predict control flow direction following conditional branches.

An attacker can train the branch predictor to choose the `taken` path on a specific conditional branch. The attacker can then provide data that would cause the `not taken` path to be chosen architecturally, but since the branch predictor is trained to predict the `taken` path, this path would be executed transiently in the wrong context.

Variant 2: Spectre-BTB

Also first introduced by Kocher et al. [3], Spectre-BTB exploits the Branch Target Buffer (BTB) in order to influence the victim to execute an attacker-chosen gadget speculatively. The BTB is used to make predictions about indirect branches. Since virtual addresses are stored in the BTB, an attacker can mistrain the branch predictor to predict branching to the address of a gadget in the victim address space by repeatedly performing indirect branches to that virtual address. When the victim process executes that indirect branch, the branch predictor will lead execution to the target gadget address, which will execute speculatively, potentially leaking confidential data to the cache.

Variant 3: Spectre-RSB

Spectre-RSB [40] [30] exploits the Return Stack Buffer (RSB) instead of the branch predictor. The RSB is a microarchitectural buffer that stores the virtual return addresses of the most recent call instructions.

Returning is a type of branching, and so the CPU tries to predict the control flow of a return instruction before it is resolved, i.e., the return address is read from memory. To make these predictions, the CPU pops the top of the RSB and uses it as its prediction.

Like other microarchitectural structures, the RSB is not shared between physical cores.

Predicting leads to speculative execution, and mispredictions are inevitable. This occurrence is most common on protection domain switches, like switching from kernel to userspace. Also, since virtual addresses are stored in the RSB, and the RSB is a per-core structure, other processes executing on the same physical core can pollute this buffer. This means that an attacker can induce misspeculation on return instructions, causing a transient path to be executed to leak data through a covert channel, like the cache.

Variant 4: Spectre-STL

Spectre-STL, first introduced by Horn [11] exploits speculation in the store-to-load (STL) forwarding logic. Speculation does not arise only from control flow predictions but also from predicting data dependencies.

A Store-to-Load dependency requires that all preceding stores that affect it must first be assured for a load to be performed. This can be a source of delay in the pipeline, so modern CPUs use a memory disambiguator to predict which loads do not need to wait on stores to be performed. A mispredicted load could then read data from the cache before the preceding store is executed and transiently perform logic with this stale value, encoding it in a covert channel that an attacker could later recover.

2.5.2 Spectre PHT

Our work focuses on Spectre-PHT. In this Spectre variant, the attacker's goal is to mistrain a bounds check condition on a conditional branch to perform an out-of-bounds memory access in a transient path.

Using listing 2.1 as a reference (from [41], an attacker could mistrain the conditional branch in line 1 by repeatedly performing the conditional branch with in-bounds values, i.e., with `x` less than `array1_size` which will train the branch predictor to predict the taken branch. Eventually, the attacker provides an out-of-bounds value for `x` which would cause the `not taken` branch to be executed. However, since the branch predictor is trained to predict the `taken` branch, this branch will be executed transiently with `x >= array1_size` which will cause an out-of-bounds memory access on `array1`.

```
1    if (x < array1_size)
2        temp &= array2[array1[x] * 512];
```

Listing 2.1: Spectre-PHT gadget

Since not all conditional branches lead to speculative execution, the attacker must create the necessary conditions for speculative execution to occur. For this to happen, an attacker can, for example, evict `array1_size` from the cache, causing a cache miss on the conditional branch, which will cause a control hazard, leading to speculative execution.

Even though an out-of-bounds memory can occur, it does not necessarily follow that data is leaked. For an attacker to leak data, this data must be encoded in a covert channel that the attacker can access. We can see this data encoding in line 2 of listing 2.1. The out of bounds access occurs in `array1[x]`.

In fig. 2.2 we can see how data can be leaked by an attacker using Spectre-PHT:

1. The branch predictor is mistrained to predict `branch taken` by being given values of `x` that are less than the value of `array1_size`. Eventually, the attacker provides an out-of-bounds value for `x`, which will cause `array1[x]` to be executed transiently, retrieving a byte from out-of-bounds memory.

2. For the sake of clarity, let us assume the data at `array1[x]` corresponds to the alphanumeric character `L`. This leaked data is then used to access the lookup array, `array2`.

3. The access to `array2`, will encode the leaked data in the cache. The attacker can now scan through all entries in `array2`, using a cache attack like Flush+Reload [8]. The array index that registers as a cache hit was the one accessed by the victim, and so the attacker retrieves the value of `array1[x]`.

This work focuses on identifying Spectre-PHT gadgets in binaries. A Spectre-PHT gadget is a sequence of instructions executed in a transient path that can perform out-of-bounds memory accesses and encode the data accessed into a cache side-channel.

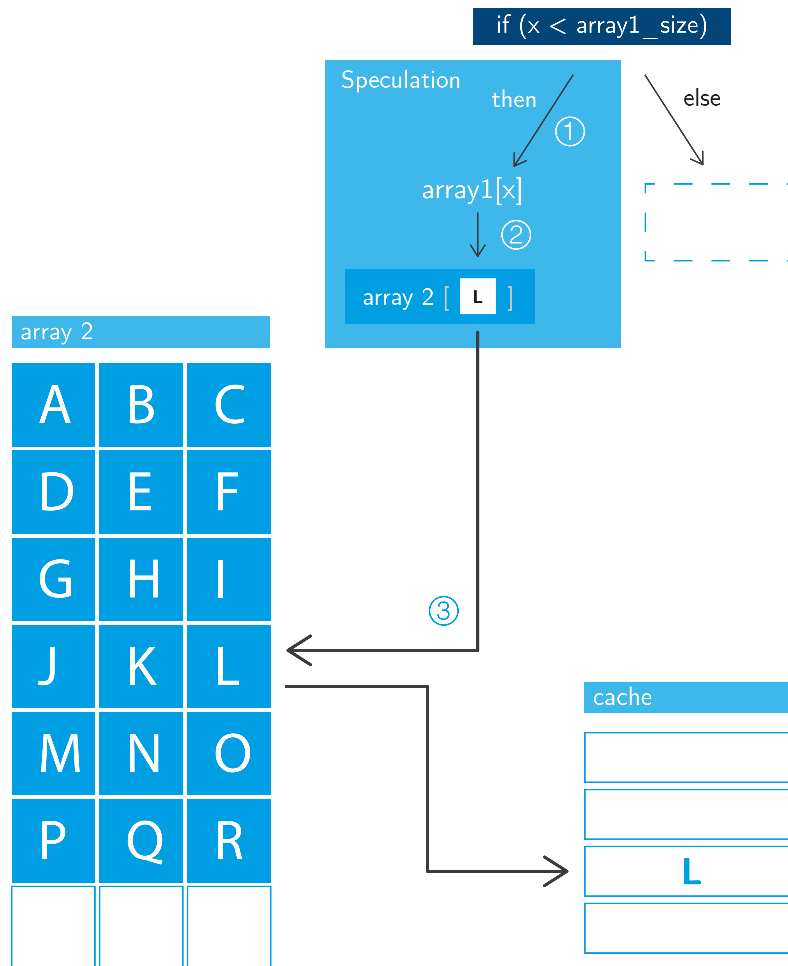

Figure 2.2: Example of encoding data into the cache using Spectre-PHT.

Examples of different Spectre-PHT gadget variants are provided by Kocher [41]. These gadget variants have been used as a litmus test for Spectre-PHT gadget detection tools.

2.5.3 NetSpectre

In 2018, Schwarz et al. [25] introduced **NetSpectre**, a fully remote Spectre-PHT attack. Alongisde a cache-based remote Spectre-PHT attack, Schwarz et al. [25] also introduce an AVX-based [42] side channel with increased performance in relation to the cache-based side channel.

Advanced Vector Instructions (AVX) is an extension to the Intel x86 architecture that provides instructions that perform operations on vectors [42, 43]. These instructions can operate on 128-bit registers and are commonly used in cryptographic algorithms [43] and for floating-point operations [42]. AVX2 was introduced to extend AVX and provide 256-bit operation support [42].

NetSpectre's [25] two main pillars are *leak gadgets* and *transmit gadgets*. NetSpectre [25] attacks perform mistraining in-place. Valid and invalid network packets are passed to the victim to train the branch predictor to mispredict branch targets leading to transient execution. [25].

To leak data, the attacker causes the victim to execute two gadgets, a *leak* and a *transmit* gadget:

```
1    if (x < bitstream_length)
2        if(bitstream[x])
3            flag = true
```

Figure 2.3: NetSpectre leak gadget. Leaks 1 bit from a bitstream based a user-provided value `x` [25].

- **Leak** gadgets speculatively access data that can be out-of-bounds and change microarchitectural state based on that data. Listing 2.3 shows an example of a leak gadget [25].
- **Transmit** gadgets perform arbitrary operations with the microarchitectural data that was tempered by the leak gadget. The runtime of this operation must be affected by the microarchitectural state being used. [25]

The gadget in fig. 2.3 can be exploited to leak data remotely as follows:

1. Send several packets where the value of `x` is in-bounds to mistrain the branch predictor. The branch predictor will then predict that the branch should be taken.
2. Send a packet with a value of `x` that is out-of-bounds, causing the taken path to be executed speculatively. This leads to the operation in line three being executed transiently depending on the value of `bitstream[x]`, causing `flag` to be brought to the cache or not.
3. Use a transmit gadget which consists of an operation whose runtime depends on the value of `flag` and measure that time to leak the value of `bitstream[x]`.

The procedure above can be followed repeatedly and with arbitrary values of `x`, leading to arbitrary memory leakage. As the differences in response time are in the scale of nanoseconds, to leak an arbitrary bit with a significant confidence level, these differences must be averaged over multiple measurements. [25]. Since these response timing differences are measured of the network and the attacker does not have access to the victim's cache, it is not possible to use the same cache attacks as in local Spectre attack variants, i.e., Flush+Reload [8], or Prime+Probe [9], a remote variant of Evict+Reload [25, 34, 36] is used instead.

AVX-based Covert Channel

Schwarz et al. [25] introduce a novel covert channel based on an AVX2-based side channel. AVX2 instructions perform operations on 256-bit registers. This side-channel exploits the fact that the CPU can power down the upper part of the AVX2 unit. [25, 44] The AVX2 unit is shut down to save power. As soon as an AVX2 instruction is executed, the unit is powered back up. When powering up, the upper half of the AVX2 unit is not available right away. Instead, there is a *warm-up period* where the AVX2 unit can still perform 256-bit operations but uses the lower half (128-bits) only, which causes slower computation times [45]. After the warm-up period, the AVX2 unit is again fully usable. The upper half of the AVX2 unit is shut down after approximately 1ms of inactivity [25, 45].

The timing difference between AVX2 instructions when the AVX2 unit is warmed-up and when it is powered down is more than twice the difference between cache hits and misses [25] which makes this

```
if (x < bitstream_length)
    if(bitstream[x])
        _mm256_instruction();
```

Figure 2.4: NetSpectre leak gadget using an AVX-based covert-channel to encode confidential data [25].

side-channel better for remote usage than traditional cache side channels. Setting up the AVX2 unit for the next leak gadget is also easier to perform than evicting data from the cache, as the attacker needs to wait for the unit to power down [25]

To exploit this side channel, an attacker can use a leak gadget like the one in fig. 2.4, which, depending on the result of the comparison in line 2 during speculative execution, executes or not an AVX2 instruction transiently. Schwarz et al. [25] showed that for AVX2 instructions, in particular the `VPAND` instruction, there is no performance loss if the last AVX2 instruction was executed less than 0.5ms ago. The transmit gadget then needs to perform an AVX2 instruction not longer than 0.5ms after the leak gadget is executed [46] to leak the result of the comparison in line 2 of fig. 2.4.

2.5.4 Spectre Countermeasures

Several Spectre attack countermeasures have been proposed by Intel [12], in the form of *Indirect Branch Restricted Speculation* (IBRS) which prevents Spectre attacks to access SGX [47] enclave memory. Another countermeasure introduced by intel is the *Indirect Branch Predictor Barrier* (IBPB) which prevents instructions executed before this barrier to modify indirect branch prediction targets executed after the barrier [12]. Intel also proposes that *process isolation* be used to make sure data cannot be leaked between processes. This approach was applied to web browsers by Google with Strict Site Isolation [48] (SSI) which separates each website's data in its own separate process. However, Agarwal et al. [49] were able to exploit SSI implementation issues to leak data from different websites from JavaScript [49].

Other mitigations targeting Spectre-PHT and other variants, proposed by Intel, ARM and AMD include the insertion of serializing instructions like `lfence` to locally disable speculative execution. However, disabling speculative execution has a negative impact on program performance [41]. There have been some proposed mechanisms for selectively inserting these serializing instructions in order to achieve a good balance of security and performance [16, 20, 23], but ultimately this solution is not optimal.

Hardware mitigations have the most potential to mitigate speculative execution attacks. However, they usually involve architecture redesign and are expensive. SafeSpec [14] and InvisiSpec [13] were hardware speculation attack countermeasures. Both SafeSpec and InvisiSpec proposes the creation of a separate structures to keep track of microarchitectural state affected by transient instrutions that are invisible to attackers, which removes the threat of microarchitectural side-channels during speculative execution. These solutions have not been implemented in the real world due to their high cost.

2.6 Meltdown Attacks

While Spectre attacks exploit branch mispredictions, Meltdown [4] attacks exploit transient instructions that are executed following an exception. In affected microarchitectures, results from faulty instructions can still be used in transient instructions before the faulty instruction is retired, and consequently, the exception raised [4]. Following transient instructions can then encode this data in a covert channel by bringing it to the cache. An attacker can then retrieve this data from the cache using a cache attack like Flush+Reload [8]. This was the first Meltdown variant introduced and is known as Meltdown-US. Other variants have since then been reported.

Foreshadow [5] is a Meltdown variant that can be used to leak data from SGX [47] enclaves [5] and can be mounted from guest systems to leak hypervisor memory [6]. Leaking SGX enclave data is done by exploiting the *present* bit in the page table to cause a page fault upon access, instead of SGX's abort page semantics [5, 47]. After the attacker caused a page fault on an SGX enclave page, data in the L1 cache can now be leaked the same way as in Meltdown-US.

2.7 Program Analysis

Program analysis is the process of analyzing programs. Program analysis can have diverse goals, such as detecting where optimizations can be implemented and tested for correctness.

Detecting security vulnerabilities has been one of the main focuses of program analysis, and it has helped to identify countless vulnerabilities over the years. However, program analysis has its limitations; one of them is that detecting security vulnerabilities is often an undecidable problem. This limitation means that automatic analysis tools usually suffer from producing false positives or false negatives. A false positive occurs when a tool detects a security violation when there is none. A false negative occurs when a tool deems a specific piece of software as secure and contains a security violation.

There are three main approaches to program analysis: dynamic, static, and hybrid analysis. These refer to the timing of when program analysis is performed.

- **Static analysis** is performed before program execution and is usually faster because it does not require the overhead of launching processes and executing the instructions being analysed. Static analysis techniques include:
  - Control-flow analysis
  - Model checking
  - Program verification
  - Taint Analysis

  Although faster, static analysis techniques do not have access to actual user input data. This often results in an increased number of false positives.

- **Dynamic analysis** is performed during program execution or after program execution by analyzing program traces or other collected data. Dynamic analysis techniques include:
  - Dynamic type checking
  - Monitoring
  - Binary instrumentation
  - Testing

  Opposite to static analysis, dynamic analysis has access to runtime information such as user inputs and control flow paths. This helps increase precision. On the other hand, it is restricted to only a subset of all possible executions. Decreasing the search space can lead to false negatives.

- **Hybrid analysis**, as the name implies, is a combination of static and dynamic analysis. Hybrid analysis tools can use the output of static analysis as the input of dynamic analysis and vice versa. An example of a hybrid analysis technique is symbolic execution.

In this work, we use a hybrid analysis approach by using the outputs of static analysis techniques as inputs to symbolic (concolic) execution.

2.7.1 Taint Analysis

Taint Analysis is a type of program analysis that is usually static but can also be performed dynamically.

There are two main concepts of Taint Analysis: sources and sinks.

- **Sources** are sources of taint for program data. For example, user input functions are often considered as taint sources.
- A **Sink** is a point in a program where tainted data should not reach.

Taint analysis is often used to verify whether information can flow from user input to a sensitive function in a software-security context.

There is a vulnerability in C programs called *format-string vulnerability*. A format-string vulnerability consists of the first argument of `printf` being controlled by the user. An attacker can leverage this control to use format-string modifiers like `%p` or `%s` to leak program data. A format-string vulnerability can even be leveraged to build a *write-what-where* primitive that an attacker can use to achieve arbitrary code execution by using the `%n` option to write arbitrary data in program memory.

We can use taint analysis to analyze C source code and look for an information flow from a user input function to a printf. Listing 2.2 shows an example of a C program containing a format-string vulnerability. We can label all user input functions as sources and printf as a sink. In this example, `gets` would be a source. After line 5 is executed, `s` is now tainted. Line 6 will propagate the taint to the `t` variable, since

now t points to s which contains tainted data. Finally, line 8 executes, which translates to a sink function being executed with a tainted argument which constitutes a security violation.

```
int main(){
    char s[20];
    char* t;

    gets(s); // gets is a source -> s is tainted
    t = s;   // t is now tainted

    printf(t); // printf is a target -> vulnerable

    return 0;
}
```

Listing 2.2: Format-string vulnerability

A tainted variable can be sanitized, which means it is no longer tainted. Take a function sanitize that removes all format-string modifiers from a string, like %p and %n. Calling this function on a tainted format string before it is given to the printf as an argument would make it impossible for an attacker to exploit its control. Listing 2.3 illustrates this example.

```
void sanitize(char* s); // removes format-string modifiers from s

int main(){
    char s[20];

    gets(s); // gets is a source -> s is tainted
    sanitize(s); // s is now sanitized

    printf(s); // printf is a target -> not vulnerable

    return 0;
}
```

Listing 2.3: Format-string vulnerability

Taint analysis can help detect Spectre-PHT gadgets by tracking whether user input can reach an out-of-bounds check on a conditional branch. If it can, the conditional branch can be mistrained, and a transient path can be executed.

2.7.2 Symbolic Execution

Symbolic execution [50] [51] uses symbolic values instead of concrete input values to represent program variables. These symbolic variables belong to the symbolic engine's symbolic state. The other

component of symbolic execution is the symbolic path constraints which is a formula that aggregates the constraints imposed on the variables by conditional statements in a given path. During execution, both the path constraint and the symbolic variables are updated. First, constraints are applied to symbolic variables based on the program's control flow and conditional statements. These constraints are later solved using a constraint solver [52] to verify path feasibility, giving possible values for the symbolic variables that satisfy the path constraints. If the path constraint is not satisfiable, then the path is not feasible.

Modern symbolic (concolic) execution performs symbolic execution dynamically. A program is run concretely, keeping a concrete state and a symbolic state at the same time. A program is started with some concrete input. As the program runs, symbolic constraints are added to those inputs when control-flow-altering statements are encountered. These constraints are evaluated, and new inputs are generated to guide execution to specific paths. In the end, the goal is to explore all possible paths, achieving maximum code coverage.

2.7.3 Fuzzing

Fuzzing [53] is a dynamic program analysis technique. It consists of testing a program by generating inputs in order to achieve maximum code coverage. When a program crashes, the fuzzer reports a bug. Additional integrity checks can be added to the program to help in detecting bugs [16, 53].

A fuzzer generates inputs, either randomly, based on an input grammar [53] or by mutating a given *input corpus*, which is a collection of program inputs. It then runs the program with those generated inputs and reports crashes or integrity check violations. In order to obtain maximum code coverage, fuzzers like American Fuzzy Lop (AFL) [54] use mutational fuzzing, where inputs are generated by mutating existing inputs. Thus, these fuzzers are referred to as mutational fuzzers.

2.7.4 Intermediate Representations

Intermediate representations [55] (IR) are program representations that are designed to abstract a program's logic, facilitating processing endeavors such as program translation or code optimizations [56]. These representations are commonly used by compilers like LLVM [57] and GCC [58] to provide cross-platform code compilation. Program analysis tools commonly use intermediate representations because they facilitate semantic analysis by equalizing syntax differences across platforms and programming languages. Abstract Syntax Trees (AST) and Intermediate Languages (IL) are two examples of IRs [55].

2.7.5 Just-in-time compilation

Just-in-time (JIT) compilation, or dynamic compilation, refers to the process of compiling code to be executed at runtime. JIT compilation serves as a time, and performance optimization [59]. Thus, JIT compilation is used to combine both benefits of static compilation and interpretation [59]. While statically compiled programs offer greater performance because they can be compiled to run machine-code

directly, interpreted programs are typically smaller but require an interpreter. However, interpreters also have access to information that is unknown before runtime, such as input parameters and control-flow [59]. Thus, JIT-compilation is typically used in interpreters, or virtual machines [60] to obtain the performance benefits of static compiled programs without sacrificing access to information about input parameters, control flow, and execution environment [59]. Instead, this information is leveraged to generate optimized machine code at runtime that is then executed for significant performance improvements.

Chapter 3

Related Work

This chapter will provide an overview of other Spectre gadget detection tools, what they provide, and their lack. In section 3.1 we will look at **SpecFuzz**, a dynamic analysis tool that uses fuzzing to identify out-of-bounds memory accesses in transient paths. Section 3.2 will provide an overview of **SpecTaint**, a recent dynamic taint analysis tool. **FastSpec**, a tool that uses neural networks to identify Spectre gadgets, will be explored in section 3.3. In section 3.4 we discuss **oo7**, a tool that uses control flow extraction, taint analysis and address analysis to detect Spectre gadgets. **Pitchfork**, **KLEESpectre**, **Spectector** and **Binsec**/**Haunted** will be explored in sections 3.7 and 3.8, respectively. These tools make use of symbolic execution to identify Spectre gadgets. Finally, we will provide an overview of how hardware performance counters (HPC) can be used to detect side-channels and Spectre attacks at runtime in Sections 3.9 and 3.10.

3.1 SpecFuzz

SpecFuzz [16] by Oleksenko et al. [16] is a tool that claims to enable dynamic testing for speculative execution vulnerabilities. It instruments a program to add speculative execution logic at compile time that simulates speculative execution behavior. Then, it fuzzes [53] this instrumented version of the program, trying to achieve maximum code coverage in order to detect out-of-bounds memory accesses in speculative execution.

SpecFuzz [16] uses a modified version of the clang [61] compiler (clang-sf) that instruments conditional branches, adding speculative execution logic at compile time to simulate speculative execution. Fig. 3.1 shows an example SpecFuzz's instrumentation. SpecFuzz then makes use of Google's Honggfuzz [62], a mutational feedback-driven fuzzer to fuzz the generated binaries to find Spectre gadgets.

SpecFuzz leverages AddressSanitizer [63] to check for out-of-bounds accesses. This check is the only one that SpecFuzz performs to increase fuzzing throughput, considering all out-of-bounds memory accesses performed in simulated speculative paths as potential Spectre gadgets. Unfortunately, this is a flawed model that is prone to false positives.

Another shortcoming of SpecFuzz is that runtime faults during simulated speculative execution halt

```
1     if x < array_size:
2         result = array[x]
3         ...
4
5
6
7
8
9         ...
```

Listing 3.1: Native conditional branch

```
1     checkpoint()
2     if x >= array_size:
3         goto skip_branch
4     if x < array_size:
5 skip_branch:
6     result = array[x]
7     ...
8     if terminate_simulation():
9         rollback()
```

Listing 3.2: Instrumented conditional branch

Figure 3.1: Side-by-side of native code and code produced by the SpecFuzz custom compiler [16]

execution and trigger a rollback. Unfortunately, this is not accurate since not all CPUs behave in this manner, as shown in the Meltdown-type attacks [4, 7] which exploit the execution of transient instructions that occur after an exception until the exception becomes architecturally visible, i.e., until the faulting instruction is retired.

Finally, SpecFuzz relies on the inputs generated by Honggfuzz and the code coverage it can produce, meaning that whether a path is explored or not is probabilistic. Ultimately, this means that paths that contain potential Spectre gadgets might not be explored by the fuzzer and thus, not detected. Furthermore, even if one of these paths is executed by the fuzzer, it might not trigger an out-of-bounds memory access, leading to Spectre gadget not being detected, i.e., a false negative.

3.2 SpecTaint

SpecTaint [23] extends a dynamic taint analysis platform (DECAF [64]) to instrument speculative execution logic at runtime to simulate speculative execution and then use a semantic-based gadget pattern to identify Spectre gadgets.

The authors Qi et al. [23] claim to outperform other state-of-the-art tools both in precision and recall while not drastically increasing runtime. Unfortunately, the tool's source code has not yet been open-sourced, so we could not verify these claims.

SpecTaint developed a taint pattern similar to the one of oo7 [20] that relies on program semantics instead of syntax. This pattern is very similar to the one used by our work since SpecTaint was released in early 2021, during the development of SpecTacle.

SpecTaint applies the taint pattern mentioned above to speculative paths in runtime, using a modified version of the DECAF [64] binary analysis platform built on top of QEMU [65], a system emulator. This modified version of DECAF adds speculative execution logic akin to that of SpecFuzz [16], i.e., the state is saved before executing a conditional branch, the opposite path of the conditional branch is then executed until it finishes or until the number of instructions in the Speculation window is reached, and finally the saved state is restored. During the simulation of speculative execution, the taint is propagated, and the taint pattern mentioned above is matched.

This approach is subject to the inherent shortcomings of dynamic analysis. For example, program path coverage may be less than 100%, meaning that not all potential Spectre gadgets are analyzed.

3.3 FastSpec

FastSpec [19] takes a unique approach to Spectre gadget detection. FastSpec claims to generate a set of over 1 million Spectre-PHT gadgets and uses them to build a classifier based on BERT [66], a novel Neural Embeddings technique. According to our research, this is the only attempt to use Natural Language Processing to build a Spectre Gadget detection tool.

To generate novel Spectre-PHT gadgets, Tol et al. [19] uses mutational fuzzing and Generative Adversarial Networks (GANs) [67] based tool. FastSpec's input consists of an assembly file. The assembly in the input file is then divided into tokens. A token is a space-separated string of symbols extracted from the input assembly file. Tokens are then distributed across windows of a configurable size `x`. Token windows overlap with consecutive windows by a fixed amount, in a *sliding window* fashion. Each window is fed to the classifier, which outputs a confidence level that there is a Spectre-PHT gadget in that window. The final output of FastSpec is a file that maps token windows to a confidence value between 0 and 1.

Since FastSpec ignores all labels in the assembly file, tokens corresponding to instructions from different but contiguous functions in memory are grouped in token windows for classification, encoding possible sequences of instructions that could not appear during program execution, which leads to false positives.

Through our analysis of FastSpec's training dataset [68], we found that some of the generated Spectre-PHT do not constitute a valid Spectre-PHT gadget. Take fig. 3.2. We can see that the memory access performed in line 10 is not guarded by a bounds check. Then follows a conditional branch which tests the contents of `array1` instead of performing the bounds check on the index that is being accessed. After the conditional branch, in line 17, a second memory access is performed where the memory contents accessed before speculation are encoded in the cache. This does not constitute a bounds-check bypass Spectre-PHT gadget because it is possible to use this code to leak out-of-bounds data architecturally without the need for speculative execution.

The fact that code like the one in fig. 3.2 is used to train FastSpec's classifier leads to false positives by design.

FastSpec authors claim to verify every gadget generated to train FastSpec by providing only out-of-bounds values to the gadgets and checking for leakage. Providing only out-of-bounds values makes it so the branch predictor is not be mistrained and so no data is expected to be leaked. If it is, then the gadget is discarded, because then it can be leaked architecturally. This verification algorithm clearly does not produce the expected results since the gadget in fig. 3.2 was not discarded, because the conditional branch does not directly depend on the provided index.

```
.text
  .file "17.c"
  .globl  victim_function     # -- Begin function victim_function_v17
  .p2align  4, 0x90
  .type victim_function,@function
victim_function:                        # @victim_function_v17
  .cfi_startproc
# %bb.0:
  leaq  array1(%rip), %rax
  movzbl  (%rdi,%rax), %eax
  cmpl  %eax, array1_size(%rip)
  jbe .LBB0_2
# %bb.1:
  cmovaeq %rdx, %rcx
  shlq  $9, %rax
  leaq  array2(%rip), %rcx
  movb  (%rax,%rcx), %al
  orq %r13, %r12
  andb  %al, temp(%rip)
.LBB0_2:
  retq
.Lfunc_end0:
# .size victim_function_v17, .Lfunc_end0-victim_function_v17
  .cfi_endproc
```

Figure 3.2: FastSpec training gadget [68].

3.4 oo7

Oo7 [20] uses taint analysis to detect Spectre gadgets. It does this by checking code against a semantic taint pattern during a transient path, simulated by *forced execution*. This pattern requires the presence of a *Tainted Branch*, a *Read Secret* and a *Leak Secret* stage.

Oo7 uses the Binary Analysis Platform (BAP) [69] which helps automate binary analysis. First, it lifts instructions to an intermediate representation, Binary Instruction Language (BIL). BAP also handles taint propagation. Then, the oo7 checks for the semantic taint pattern, flagging every match as a potential Spectre gadget.

The main disadvantage of oo7 is that it is reported to have very high runtimes, limiting its real-world usage. Because it performs static taint analysis, it is also prone to overtainting, which leads to false positives. In the FastSpec [19] paper, it is also reported that oo7 fails to detect gadgets that use *cmov*, *xchg* or *set* instructions, leading to false negatives.

3.5 Pitchfork

Pitchfork [70] is a tool introduced by Cauligi et al. [70] that uses symbolic execution to validate *speculative constant-time* semantics, which they also define. Speculative constant-time is an extension of the *constant-time* paradigm [71] that is used across modern cryptography to deal with timing side-channels, by ensuring that computation time does not depend on confidential data. Speculative constant-time extends this idea to speculative execution which enables for microarchitectural side-channels to be dis-

covered. Pitchfork can check for speculative constant-time violations by modeling a three-stage machine that supports out-of-order and speculative execution.

Pitchfork defines a set of semantic rules that map *speculative constant-time* properties to their machine model and check for violations to those semantic rules during symbolic execution. Ultimately, Pitchfork suffers from the problems inherent to symbolic execution, such as path explosion that affect its scalability to large codebases.

3.6 KLEESpectre

KLEESpectre [72] by Wang et al. [72] uses symbolic execution to detect Spectre gadgets by modeling transient execution during symbolic execution. KLEESpectre models speculative execution symbolically by forking execution into four paths at each conditional branch, two regular paths, and two transient paths. The regular paths represent the regular *branch taken* or *branch not taken* paths, while the transient paths model transient execution switching the context of the previous paths, i.e., the *branch taken* path is executed with program state that would normally lead to a *branch not taken* path, and vice versa.

KLEESpectre [72] provides *symbolic cache modeling* to detect cache side-channel leaks through speculative execution paths. This platform is built on top of KLEE [73] and analyses LLVM [57] bytecode. By forking execution into four paths at each conditional branch, KLEESpectre accentuates the problem of path explosion already imposed by traditional symbolic execution approaches, leading to high execution times.

3.7 SPECTECTOR

SPECTECTOR [17] leverages symbolic execution to detect Spectre gadgets in x86-64 programs. SPECTECTOR verifies if programs follow *Speculative non-interference* (SNI). SNI states that two initial program states can only be distinguished under speculative semantics if they can also be distinguished under non-speculative semantics.Non-speculative semantics refer to standard program execution, while speculative semantics follow the execution of mispredicted branches to simulate speculative execution.

SNI is a novel semantic property for detecting Spectre attacks and was introduced by Guarnieri et al. [17]. SPECTECTOR uses symbolic execution to derive a representation of the memory accesses along execution traces and branch targets along all possible paths of a program, which leads to an SMT (Z3 [74]) formula with the conditions that two initial program states produce the same access patterns in speculative and non-speculative semantics. If this formula is satisfiable, then SNI is ensured, and the program is secure. Otherwise, a program is considered to have memory leaks in speculative execution.

One drawback of this SPECTECTOR is that only the x64 Instruction Set Architecture is supported. A second drawback is that x64 is not fully supported, although the Guarnieri et al. [17] claim that providing this support is trivial. The fact that there are unsupported instructions makes it hard to evaluate SPECTECTOR for real-world applications. Since unsupported instructions are skipped, this can also lead to

false positives and false negatives when identifying Spectre-PHT gadgets.

3.8 Binsec/Haunted

Binsec/Haunted [18] expands on the works of Cauligi et al. [70] and Wang et al. [72]. It uses symbolic execution to verify *speculative constant-time* but tackles a significant drawback of symbolic execution, which is *state explosion*.

Binsec/Haunted applies Relational Symbolic Execution (RelSE) to verify *speculative constant-time* semantics in symbolic execution. To model symbolic execution, Binsec/Haunted uses a modified version of the explicit approach presented by KLEESpectre [72] that executes transient paths simultaneously with regular paths to reduce the number of states generated. Daniel et al. [18] named this technique *Haunted RelSe*. Haunted RelSE works by simplifying the symbolic expressions to evaluate speculative constat-time with a simpler formula that explores fewer paths and adds fewer constraints. This optimization makes it possible to explore a speculative path that encompasses both transient and regular behavior, reducing the number of states needed during symbolic execution.

Haunted RelSE is implemented on top of an existing tool called Binsec [75], a binary-level analyzer, resulting in a new tool called Binsec/Haunted.

3.9 Speculator

Speculator [21] takes a dynamic approach to detecting speculative execution attacks (SEAs). This approach is applicable to other classes of SEAs besides Spectre [3, 11, 25], like Meltdown [4–6] variants [21]. Speculator proposes using speculative execution markers, particularly instructions or sequences of instructions that are detectable by performance counters even when executed in transient paths and therefore discarded. These performance counters are architecture-specific.

Performance counters that make good markers are ones that track *issued* or *executed* instructions, regardless if they were retired or not [21]. Counters that track only specific instructions like `lea` or `div` instructions also make for good speculation markers [21]. Speculator takes as input a snippet of code to be analyzed and appends speculative execution markers to it. If Speculator detects these markers, then it concludes that the CPU executed the snippet.

Speculator can be used to detect transiently executed instructions and determine speculation window sizes, among other microarchitectural properties. Although this tool is not used to search through codebases for possible Spectre gadgets, it has been demonstrably used to debug, evaluate and prototype Spectre-type attacks [21].

3.10 Dynamic Process Isolation

Recently, Schwarzl et al. [22] proposed a novel probabilistic Spectre attack detection technique based on HPC that can be used during runtime and imposes a low overhead of around 2%t performance decrease [22].

Two approaches to Spectre attack detection were proposed by Schwarzl et al. [22]. The first approach uses *Precise Event-Based Sampling* to recover the number of mispredicted and retired branches. These samples are then compared, using probabilistic methods, to the distribution of a template attack [22]. For example, to generate a distribution of the mispredicted branches for Spectre-PHT attacks, a Spectre-PHT gadget with in-place mistraining [3] was executed for a fixed amount of time, and the performance counters were sampled. This template attack showed a misprediction distribution pattern that could be easily distinguished from a benign program. This information can then be used to make probabilistic claims about a given branch misprediction distribution to distinguish benign from malicious programs [22].

The second approach is similar to an approach proposed by Gruss et al. [39] to detect cache attacks. This approach, which proved to be faster, samples the number of retired branches but proceeds to normalize it with the number of *instruction translation lookaside buffer* (iTLB) accesses. First, different performance counters are collected, like the number of iTLB cache misses, the number of branch instructions, and the number of cache misses. These counters are then normalized using the number of iTLB accesses, which relates the counter values and the code size. Since Spectre attacks have a small code footprint, it is possible to calculate a threshold that enables the distinction between malicious and benign code [22].

Chapter 4

Implementation

In this chapter, we will describe SpecTacle's implementation choices and details. Section 4.1 will provide an overview of SpecTacle's architecture, its main components, and how they interact with eachother. Sections 4.2, 4.3, 4.4, 4.5 and 4.6 go into detail about each of SpecTacle's components. Finally, Section 4.7 discusses software optimizations implemented in SpecTacle, as well as others left as future work.

4.1 High-level Overview

The core of the tool consists of three components: the `Explorer`, the `Executor` and the `Validator`. Their interaction is represented in fig. 4.1. In this section we briefly describe these components leaving a more detailed description of each component to the subsequent sections.

4.1.1 Pre-processing and Locating Conditional Branches

During the pre-processing stage, we provide the necessary support for the analysis of JIT-compiled languages, as well as integration with FastSpec [68] output. This stage's goal is to produce a binary that can be further analyzed by the remaining components of SpecTacle.

Other Spectre-PHT gadget detection tools [16–19, 23] focus on finding gadgets in compiled binaries. To the best of our knowledge, there are no tools for detecting Spectre-PHT gadgets that work with JIT-compiled languages. To make a stride in this direction, we implement primitive support for detecting Spectre-PHT gadgets in Java programs. This support is external to the core modules of SpecTacle and can be extended, in future work, to support more JIT-compiled languages. This support is discussed in Section 4.2.

Alongisde the support for Java programs, we also support integration with FastSpec [19]. Tol et al. [19] claim that FastSpec is most effective when used as a first stage in Spectre gadget detection due to its low execution times, and low false negative rates when looking at confidence levels of 0.48 or below [19]. We provide support for FastSpec output in SpecTacle by using a slightly modified version

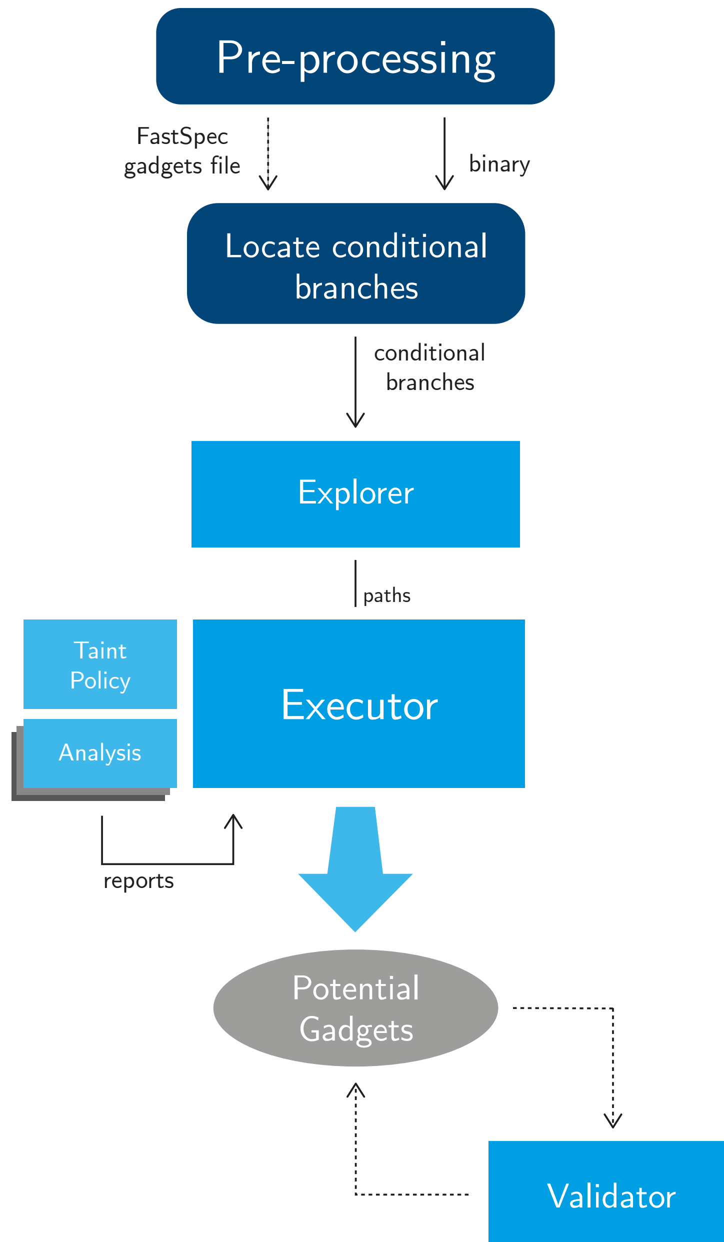

Figure 4.1: High level schematic of SpecTacle's execution flow

of FastSpec's open-source code [68] that retrieves function labels alongside the confidence levels for each token window. We then only analyze the functions that FastSpec considers vulnerable according to a given confidence threshold. Detailed information about FastSpec's output and how we achieve this integratoin is provided in Section 4.2.

As illustrated in fig. 4.1, SpecTacle requires a binary as input. After this binary is provided, the setup stage starts where the binary's conditional branches are located and given to the `Explorer`. If `FastSpec`'s output is provided, only the functions containing token windows with an assigned confidence level above the given threshold are processed, otherwise all identified functions are analyzed. Details on the conditional branch location is also be explored in Section 4.2.

4.1.2 Explorer

The `Explorer` is responsible for identifying the paths where potential gadgets are located. It returns all possible execution paths starting at the beginning of the basic block a given conditional branch belongs to and up to a configurable length. These are the execution paths where speculative execution can potentially be triggered, and consequently, a Spectre[3] gadget can be found. From now on, they will be referred to as `traces`. This module is also responsible for lifting code basic blocks present in the traces to the Vex Intermediate Representation [76]. Implementation details are provided in Section 4.3.

4.1.3 Executor

The `Executor` is the main module of SpecTacle and is responsible for processing the traces retrieved by the `Explorer`. It iterates through every Vex statement in the given traces and reflects its expected effect on guest state on the `Executor`'s own state. Vex maps each instruction to one or more Vex statements, and these statements can contain Vex expressions. Vex expressions represent operations without side effects, such as arithmetic operations, constants, memory loads.

As illustrated in fig. 4.1, the Executor contains a `Taint Policy` property, as well as one or more Analysis Hooks which are used to perform logic on the Vex statements being visited. It performs `Taint Analysis` on the Vex statements by propagating taint information according to the given taint policy. The Taint Policy property is detailed in Section 4.4.2. The Analysis Hooks can perform arbitrary analyses using the `Executor`'s state, including the taint policy. The Analysis Hooks implemented in SpecTacle are where Spectre gadget detection logic is implemented. This module is further describer and explored in Section 4.4.

4.1.4 Validator

The `Validator` takes the potential gadgets identified by the `Executor` and uses `Symbolic Execution` to check whether such a path is feasible given the constraints imposed by the conditional branch. Spectre gadgets require transient instructions to be executed during speculative execution [3], which in turn implies that a branch misprediction occurred. We apply symbolic constraints imposed by conditional branches to program state in order to simulate such branch mispredictions and then check for path feasibility given those constraints. We use angr's [77] symbolic execution engine to follow the traces that are flagged by the Analysis Hooks while gathering path constraints, as well as constraints to other variables of the symbolic sate. If these constraints are satisfiable, the Validator deems the path feasible. This module's implementation is detailed in Section 4.6.

4.2 Pre-processing

In this section, we will discuss Java code analysis and how we support it in SpecTacle. In the second part of this section, we discuss how SpecTacle provides support for FastSpec [19] integration. Finally,

in the last part of this section we discuss how SpecTacle locates conditional branches, which are then passed to Explorer for trace extraction.

4.2.1 Java Code Analysis

In order to add support for Java[78] programs we implemented `jparse` as a proof-of-concept and defined according to algorithm 1. The retrieved code contains branch instructions, like jumps and calls, to hardcoded addresses. These branch instructions are not compiler-friendly since during compilation time the compiler might not map the branch targets to the same addresses they are mapped to during execution, which leads to compilation errors. To solve this problem, we insert labels before branch targets within the bounds of the method being parsed. Jumps and calls to out-of-bounds addresses are handled by creating mock routines that perform a return instruction. This solution is not optimal and can lead to false negatives. It is possible to build a more accurate solution by analyzing the JVM process' address space and keeping context about the JIT-compiled methods, which is left as future work.

The *disassembleMethod* and *gatherMethods* represent wrappers to JRE commands. The commands used to gather the compiled methods and to disassemble those methods can be found in listing 4.1 and 4.2 respectively.

After a valid assembly file is generated by *jparse* it can be compiled and analyzed by SpecTacle.

```
java -Xbatch -XX:-TieredCompilation -XX:+PrintCompilation
-cp <classpath> <main class>
```

Listing 4.1: Gather compiled methods

```
java -Xbatch -XX:-TieredCompilation -XX:+UnlockDiagnosticVMOptions
-XX:CompileCommand=print,*{<method>} -cp <classpath> <main class>
```

Listing 4.2: Print method's JITed code

Algorithm 1: Generating compilable assembly from JVM JITed code

```
1  Function main(className):
2      methods ⟵ gatherMethods(className);
3      foreach m ∈ methods do
4          labelCounter ← 0
5          routineCounter ← 0
6          assembly = disassembleMethod(m);
7          foreach instruction ∈ assembly do
8              if instruction is a jump then
9                  label = ".LB" + labelCounter;
10                 labelCounter ⟵ labelCounter+1;
11                 instruction.replace(jump_target, label);
12                 assembly.addLabel(label);
13             end
14             else if instruction is a call then
15                 label = "_sub" + routineCounter;
16                 routineCounter ⟵ routineCounter+1;
17                 assembly.addSubRoutine(label);
18             end
19         end
20         writeFile(template(m));
21     end
```

4.2.2 FastSpec Integration

Tol et al. [19] propose that taint analysis or symbolic execution tools can be used to further analyse the potential gadgets detected by FastSpec to build an efficient end-to-end solution for finding Spectre gadgets. With this goal in mind, we provide support for analysing functions labeled as vulnerable by FastSpec in SpecTacle.

The output of FastSpec [68] consists of a file containing the confidence level for each window of tokens. A token is a space-separated string from the disassembly of the binary being analyzed. For example, the instruction `xor %ebp , %ebp` is separated into the tokens: `"xor"`, `"%ebp"`, `","` and `"%ebp"`. These tokens are grouped in windows of a fixed size and written to a `.tsv` file, one window per line. Tokens from contiguous windows overlap by a fixed amount. This way, consecutive windows contain the tokens from the previous window shifted left by x, where $x = window_size - overlap$ followed by the next token from the disassembly of the binary.

FastSpec reduces the dictionary of tokens by replacing certain types of tokens, like function labels

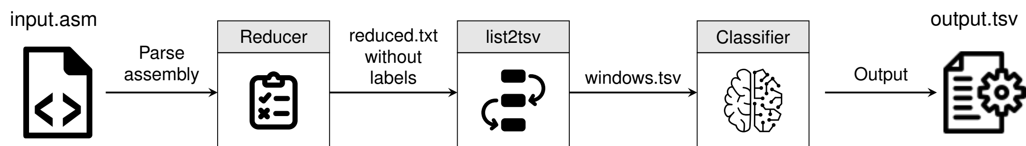

Figure 4.2: FastSpec's Execution flow

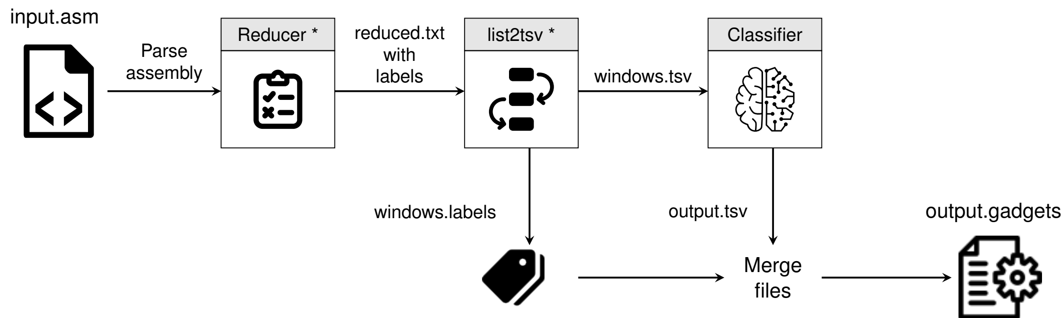

Figure 4.3: FastSpec's execution flow modified to retrieve the function labels and generate a *gadgets* file. Labels marked with * represent modified script files.

and immediate values, by generic representations, such as `<label>` and `<imm>` respectively. As we are interested in mapping the confidence levels given by FastSpec to functions and consequently memory addresses of the conditional branches inside those functions, we need to recover these labels. Fig. 4.2 illustrates FastSpec's execution flow, from the assebmly input file until the generation of the `output.tsv` file that maps each token window to a confidence level.

To get the function labels out of FastSpec, we need to modify the source code. First, FastSpec's [68] `reducer.py` script is responsible for replacing function labels, immediate values and unknown artifacts present in the input assembly file for generic tokens (`<label>`, `<imm>` and `<UNK>`). We modify this script to keep function label tokens instead of replacing them with the `<label>` token. We do this by matching the processed tokens to the following regular expression, which corresponds to how objdump identifies function labels: ([0-9a-f])+<([a-zA-z0-9_@.])+>. These label tokens are then passed to `list2tsv.py`, which will generate the `.tsv` mentioned previously. Each token in a window is assigned a function label, which is logged to a separate `.labels` file. These labels are then removed so that they are not present in the final `.tsv` not to influence the results of FastSpec's classifier.
After feeding the `.tsv` to the classifier, we match the output of FastSpec, i.e., the confidence level of each window, to its corresponding labels using the `.labels` file. From this matching we generate a `gadgets` file that contains pairs of the form `<function label, confidence level>`. The `gadgets` file along with its corresponding binary are then fed to SpecTacle. First, the `gadgets` file is filtered for function labels where the confidence level exceeds a given threshold. SpecTacle then only analyses the functions where the confidence is above that threshold. Fig. 4.3 shows the modified execution flow, with the modified script files marked with a * character.

4.2.3 Locating Conditional Branches

A key stage of SpecTacle's workflow is locating the conditional branches since our searches will always start from a basic block containing a conditional branch. To locate these branches, SpecTacle uses radare2 [79] to disassemble the binary. Then, every instruction of every function identified by radare2 is iterated over, and its operation type is checked. The operations with type `cjmp` are conditional jumps and are saved in a list.

4.3 Explorer

The `Explorer` is responsible for identifying the paths where potential gadgets are located. The Explorer starts by lifting instructions to an intermediate representation (IR), VexIR [76]. Working with an intermediate representation makes it easier to reason about program semantics because we are no longer exposed to program syntax, making SpecTacle language agnostic.

It works in three main stages:

- **Stage 1: Lift instructions to intermediate representation (PyVEX).** Each call to the `Explorer`'s `generate_paths` method is passed the address of a specific conditional branch from which to start the exploration. Every basic block that is reached by the Explorer is lifted to `Valgrind`'s [80] `Vex` [76] intermediate representation through a `python` libvex wrapper PyVEX [81], using the `angr` [77] framework. These basic blocks comprised of Vex statements are called `Intermediate Representation Superblocks` (IRSB). These blocks are a form of extended basic block [82, 83]. A *basic block* is a sequence of instructions that ends in either a branch instruction or an instruction label [55]. An *extended basic block* contains only a single entry but can contain more than one conditional branch instructions [82]. In figs. 4.4 and 4.5 we can see that the IRSBs recovered by PyVEX (fig. 4.5) have a 1-to-1 correspondence to the assembly basic blocks identified by and Capstone [84] (4.4).

- **Stage 2: Build a jump graph.** PyVEX matches each binary's basic block to an IRSB. An IRSB contains a `jumpkind` field that holds the type of the IRSB's default jump, which are the branching instructions at the end of each IRSB. *Boring* jumps correspond to conditional branches with a `branch taken` and a `branch not taken` address where execution can flow to, where all other kinds of jumps, like *return* or *call*, can only direct execution to one specific address.

  All possible jump targets are mapped to an IRSB. The target's address is added to the graph as a node and connected to the previous IRSB through a directed edge. These paths are explored recursively until either the maximum number of instructions per path is reached or until there are no more possible jump targets.

- **Stage 3: Extract paths from the graph.** After the graph is built, all possible paths that start on the first IRSB and end on each of the graph's childless leaf nodes are found through a depth-first search and returned.

The paths returned by the `Explorer` will be referred as `traces` from now on. They are represented by the `BlockTrace` class, and besides the trace, they contain the function address where they start and the address of the first branching instruction in the trace. Each `BlockTrace` is then processed by the next module of SpecTacle, the `Executor`.

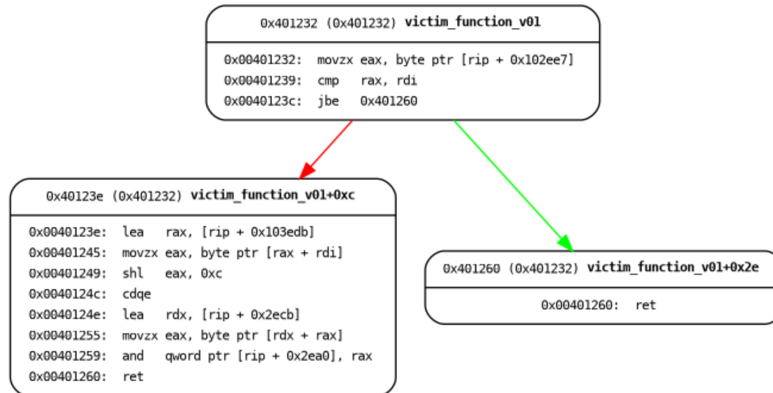

Figure 4.4: Spectre-PHT gadget CFG using basic blocks recovered by Capstone [84]

4.4 Executor

The `Executor` is the main module of SpecTacle and is responsible for processing the `BlockTraces` returned by the Explorer. The `run` method is the Executor's primary interface with user code. It receives a set of `BlockTraces`. Execution starts at the first address in the trace, which corresponds to the branch's address that will potentially trigger speculation.

Regarding the Executor's internals, it follows the Visitor [86] design pattern, implementing a `visit` method for each type of `Vex` statement found in an IRSB. A Vex statement contains a `tag` which is a string that contains the statement type's identifier. This allows the `Executor` to distinguish between different statement types and call the correct `visitor` method, as illustrated in listing 4.4. During execution, each block in the `BlockTrace` is lifted, and each `Vex` statement passed to its corresponding visitor. The Executor has internal structures that keep information on registers, temporary variables, and memory writes. These correspond to python dictionaries that map register and variable names, as well as memory addresses to a numeric value. Additionally, the taint policy and the analyses can add information to entries in these structures.

Being able to distinguish between statement types is necessary to be able to call the correct taint policy methods, as discussed in Section 4.4.2. The most relevant statement types are:

- **IMark** – Since an assembly instruction can be represented as a sequence of `Vex` statements, it is necessary to keep track of where an assembly instruction ends, and the next one begins. An **IMark** is placed before the start of a new assembly instruction and keeps track of its original address.

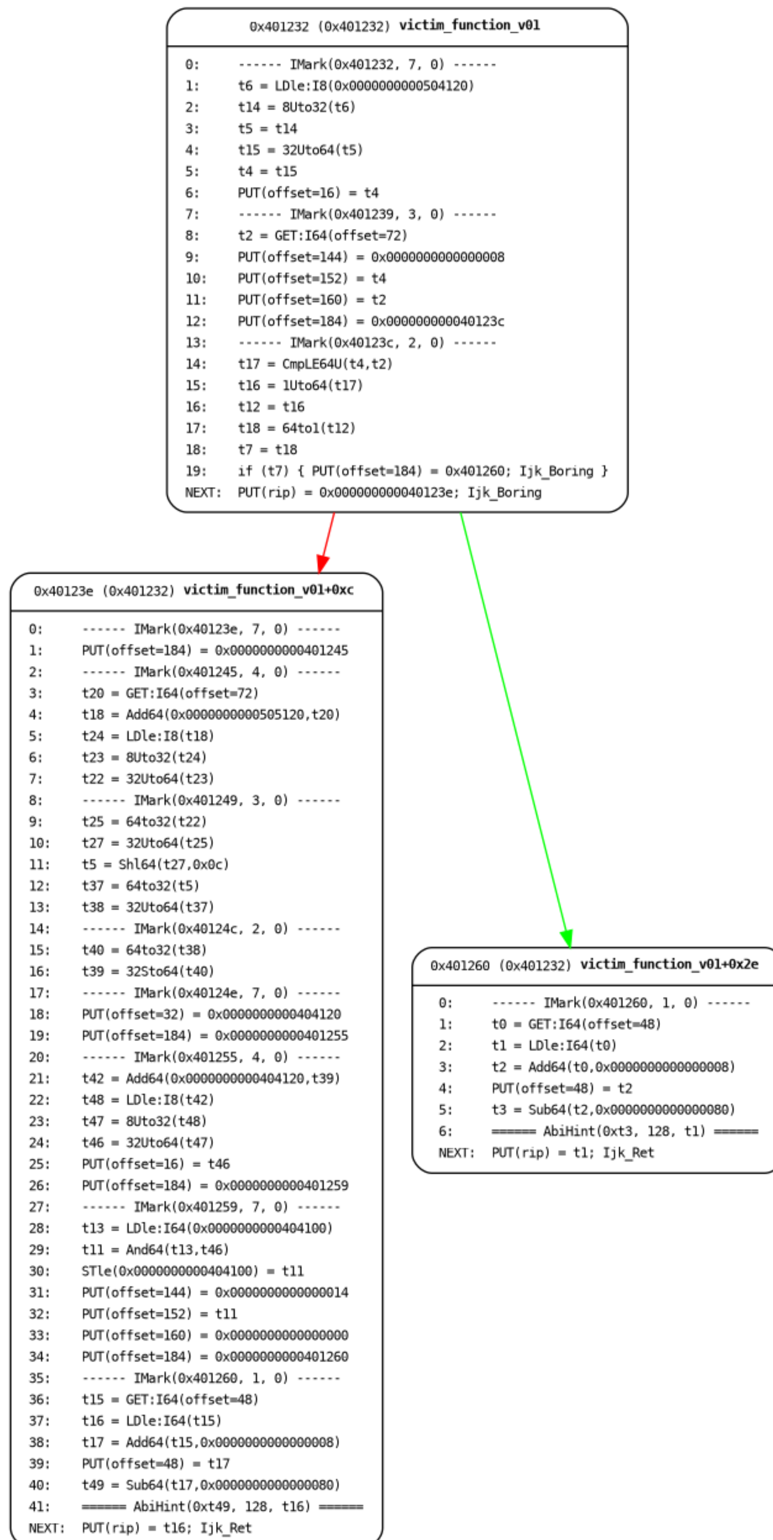

Figure 4.5: Spectre-PHT gadget CFG using IRSBs recovered using angr [82] and PyVEX [85]

- **Put** – Represents writing data into a register identified by its byte `offset` property, and a type which represents the size of the value being written.
- **Dirty** – Some statements that Vex does not directly support are handled by dirty helpers that emulate their functionality. Dirty statements identify such helpers. An example of an instruction

that is represented by a Dirty statement is cpuid.

- **WrTmp** – Represents instructions that write to `Vex` temporary variables. **WrTmp** is characterized by its `data` property which is a `Vex` expression. A notable `Vex` expression is `Get` which is the counterpart to `Put` in the sense that it represents reading data from a register, as opposed to reading data from a register.
- **Store** – A `Store` statement represents a memory write.
- **CAS** – This statement identifies an atomic Compare-and-Swap operation, either single or double-element.
- **Exit** – Represents a conditional exit from the middle of an IRSB. An Exit statement is used to represent a conditional branch.
- **PutI and GetI** – These are necessary to describe areas of the guest state where registers are indexed like in an array, like in the case with SPARC [87] register windows, for example. Vex also uses these two statement types to represent floating-point operations involving AVX [42] registers.

Executor State

Each `visitor` method simulates the statement's intended changes to the guest state and reflects them in the `Executor`'s state. The first time a register or a memory address is read from, the value is unknown to the Executor, so default values are used. Whenever a `WrTmp` statement is visited, a temporary variable might be created. This variable is identified by an integer, and the Executor uses this identifier as the key to an internal map that links variable identifiers to their value. This value can be updated through other `WrTmp` statements including arithmetic operators.

`Store` and `Load` instructions are used to read and modify a given memory address. This address can be provided as a constant or as a temporary variable. Both alternatives provide an address for the memory operation that is consistent with the Executor's state for the given trace. In the case of the address provided by a temporary variable, this can be the result of arithmetic operations, either over a constant value attributed to the variable in an earlier `WrTmp` statement or over the default value for uninitialized data. According to our observations, memory is addressed consistently throughout each trace, so this method does not commonly produce two mappings for the same memory address.

Run Method

Listing 4.3 illustrates a stripped-down version of the `run` method. First, a dictionary structure is prepared to be returned by creating an entry for each analysis installed in the Executor, as shown in line 2. Then, for each provided trace, the Executor's state is reinitialized. Reinitializing Executor state means that registers are set to their default values, and the memory and temporary variables structures are both cleared. A new taint policy is instantiated in line 5 so that all of its structures are also cleared. Lines 7 to

10 correspond to the main loop of the `run` method. For each basic block address of the given trace, the corresponding IRSB is recovered, and its statements are visited sequentially, using the `visit` statement illustrated in listing 4.4. This visit method calls the Executor's specific statement visit methods, which enforce the taint policy for the necessary statements. A detailed explanation of the taint policy is provided in Section 4.4.2. The visit method also invokes the installed analysis hooks that perform arbitrary analyses using the Executor's state. These analysis hooks generate reports which are then retrieved in lines 12 to 15 of Listing 4.3. Finally, these analysis hooks and the Executor's state are reset for the next trace.

```
1 def run(traces):
2     report = {a.tag : set() for a in self.analysis}
3     for trace in traces:
4         self.initialize_state()
5         self.policy = self.policy_type()
6
7         for block in trace.basic_blocks:
8             irsb = get_IRSB(block)
9             for stmt in irsb:
10                self.visit(stmt)
11
12        for a in self.analysis:
13            if a.report():
14                report[a.tag].update(rep)
15            a.reset()
16
17        self.reset()
18
19    return report
```

Listing 4.3: Executor's `run` method. Iterates over a set of traces, visits the IRSB statements and returns analysis reports

4.4.1 Analysis Hooks

Analysis hooks are used to reason about the Executor state and about the Vex statements that are visited. These are where SpecTacle implements its Spectre gadget detection logic. This section will provide an overview of analysis hooks and how they interact with the Executor.

When the Executor is instantiated, `analyses` can be installed in the `Executor`. An `Analysis` is a class that derives from the `DefaultAnalysis` class which contains `visitor` methods for all `Vex` statements types, which we call hooks. These methods have access to all of the `Explorer`'s internal structures and properties. In SpecTacle, there are three main `analyses` that are registered in the main `Executor` de-

pending on SpecTacle's input options: `DefaultCacheEncodingAnalysis`, `ComparisonResultAnalysis` and `NetSpectreAnalysis`. These will be explore in detail in the Gadget Detection Analyses section.

Akin to the `Executor`'s visitor methods, the name for the correct hook for the given statement type is also calculated from extracting the `PyVEX`'s statement tag and appending it to the string "hook_". In Listing 4.4 we can see the code that performs the hook calls.

```
1 def visit(self, stmt):
2     name = "visit_{}".format(stmt.tag.split("_")[1])
3     hook_name = "hook_{}".format(stmt.tag.split("_")[1])
4 
5     fn = getattr(self, name, None)
6     hooks = [getattr(a, hook_name, None) for a in self.analyses]
7 
8     if fn is not None:
9         fn(stmt)
10        for hook in hooks:
11            hook(self, stmt)
12    else:
13        raise NotImplementedError(f"Uninplemented Instruction: {stmt}")
```

Listing 4.4: Executor's statement visit method

4.4.2 Taint Policy

The purpose of the taint policy is to keep track of taint propagation The `Taint Policy` is implemented in the likes of the `Executor`, also following the `Visitor` design pattern, but it is treated as an `Analysis` by being called by the `Executor` on every statement visited. Unlike an `Analysis`, the `Executor` must always have a `Taint Policy` instantiated.

DefaultTaintPolicy

In the case a `Taint Policy` subclass is not provided at the moment of the `Executor`'s instantiation, the `DefaultTaintPolicy` class is used. The DefaultTaintPolicy's implementation will now be described. To help with the discussion, we define the following concepts:

- t(v) denotes the taint value of the entity `v`
- T/S denote Tainted and Sanitized respectively
- R_i denotes register `i`
- V_i denotes variable `i`

- m(i) denotes memory content at address `i`

+

A `Taint Policy` needs to be defined for the statement types `Put`, `PutI`, `Store`, `WrTmp`, `CAS` and `Dirty`, since only the operations performed by those statement types can propagate or sanitize taint values. Each of these statements can be summarized in an assignment of the form $lhs = rhs$, `rhs` and `lhs` standing for right-hand-size and left-hand-size of an assignment operation. In the DefaultTaintPolicy, taint is propagated as follows:

Dirty

By definition, this statement type sanitizes the `lhs`, which is a `PyVEX` temporary variable, so we can describe it as: $t(lhs) = S$

Put/PutI

The algorithm for `Put` and `PutI` is the same. These statements write to registers, either directly or by indexing in the case of `PutI`. The taint calculation for a Put statement of the form `Put(offset, rhs)` can then be represented as the following: $t(R_{offset}) = t(rhs)$.

Compare and Swap (CAS)

This statement cannot be precisely defined in the context of SpecTacle's Executor. Therefore we only consider the step which is constant and that we can predict and evaluate, which is that the temporary variable that holds the value to be tested, denoted as `old` is always populated with the contents of the memory address to be compared, as stated in `LibVEX`'s documentation [88]. So, we can interpret this statement as another $lhs = rhs$ type of assignment where lhs is the `old` temporary variable, and rhs is the content of the memory address.

Store

As this statement type represents memory writes, it is clear that its `lhs` represents the memory at the target address, and the `rhs` represents the content to be written.

WrTmp

This statement type holds most of the DefaultTaintPolicy's logic. The `lhs` in these statements is always a temporary variable, but the `rhs` can be any of the following PyVEX expressions:

- **Get and GetI** instructions read the content of a register into a temporary variable. The `rhs` in this case then is the content of the register.
- **Multi-operand operations and CCall** statement types are handled by the DefaultTaintPolicy in the same way. Its operands or arguments are considered the `rhs` of the taint propagation. The

elements of the `rhs` are iterated over, and if at least one of them is tainted, the `lhs` is considered tainted. In case more than one element in `rhs` are tainted, the last one to be iterated over is chosen to be the taint source. This heuristic serves only for code simplicity, and its improvement is left as future work. The taint propagation logic is illustrated in listing 4.5.

```
1 CCall/Unop/Binop/Trip/Qop(tmp, args):
2     for arg in args:
3         if t(arg) == T:
4             rhs.append(arg)
5         else:
6             rhs.insert(0, arg)
7     t(V_tmp) = t(rhs[-1])
```

Listing 4.5: Multi-operand taint propagation logic implemented by the DefaultTaintPolicy

- **If Then Else (ITE)** - The statement represents the ternary conditional operator composed of a `condition`, a `true` and a `false` fields. The DefaultTaintPolicy does not evaluate the condition. Instead, it takes the taint value of the `true` and `false` fields, sanitizing the lhs if they are both sanitized and tainting it otherwise.

- **RdTmp** - This statement represents a simple assignment between two temporary variables V_1 = V_2. This is the simplest case for the taint policy and is represented as follows: $t(V_1) = t(V_2)$.

- **Load** - A Load statement can propagate taint in two ways: either the content of the memory address being loaded is tainted, or the address itself is user-controlled and, therefore, tainted. The DefaultTaintPolicy prioritizes the latter, checking first if the address is tainted, in case it comes from a temporary variable, and only if this variable is sanitized the memory contents are checked. The taint propagation logic is illustrated in listing 4.6.

```
1 Load(tmp, addr):
2         t(V_tmp) = (t(addr) == T ? T : t(m(addr)))
```

Listing 4.6: Load taint propagation logic implemented by the DefaultTaintPolicy

Internally, the DefaultTaintPolicy holds two dictionaries: one for registers and one for variables. The `register` dictionary keeps a (taint, source) pair for each register in the architecture. These are initialized to ("T", None) and are updated when `Put` or `Get` statements are visited. The `variables` dictionary maps a variable to a register present in the `registers` dictionary, and is updated when visiting `Dirty` and `WrTmp` statements. When a statement modifies a variable's taint value, by design its taint source is a register and so its entry in the `variables` dictionary will point to that source register. There is a particular entry in the `registers` dictionary called `const` which is never tainted and is pointed to by variables that have been sanitized by a constant.

The visitor method for `Store` statements behaves differently. Instead of updating the DefaultTaintPolicy's internal strucutres, it alters the `Executor` 's memory model directly, updating the `taint` field the corresponding memory entries being modified.

StrictTaintPolicy

To reduce the number of false positives and increase SpecTacle's precision potentially at the cost of some false negatives, we implemented a stricter taint policy (StrictTaintPolicy) for testing. This taint policy only introduces taint to registers or memory addresses encountered before the speculation started. We observed that the majority of the gadgets found during SpecTacle's evaluation performed some operation with the *attacker-controlled* data in the basic block where the conditional branch is located. This taint policy exploits this fact to reduce the number of tainted memory loads identified and consequently reduce the number of flagged gadgets, the majority of which were false positives.

4.5 Gadget Detection Analyses

This section will describe the 3 main gadget detection analyses used by SpecTacle: DefaultCacheEncodingAnalysis, ComparisonResultAnalysis and NetSpectreAnalysis.

A generic analysis that derives from `DefaultAnalysis` is expected to produce a `Report`. Reports contain a trace, a function address, and a branch address. These fields serve to keep context about specific areas of the binary.

The goal of gadget detection analyses is to produce a report containing Spectre gadget candidates, represented by the class `GadgetCandidate` that inherits from `Report`. In addition to the fields of a generic Report, GadgetCandidates also contains a list of memory accesses. SpecTacle uses the `MemoryLoad` class to represent these accesses. A MemoryLoad represents a Vex Load statement encountered when analyzing a given trace and holds information about where those loads are performed. It also keeps information on the taint source and the value of the address being accessed.
For example, the string representation of a `MemoryLoad` at address 0x401240 that writes to the temporary variable $t24$ and is tainted by the rdi register is as follows: `0x401240 -> var:t24 ; by:rdi`.

There can be several types of reports. A report contains the relevant data for further analysis and verification of each gadget type being identified. Each analysis type results in a different type of report:

- DefaultCacheEncodingAnalysis - GadgetCandidate
- ComparisonResultAnalysis - GadgetCandidateV10
- NetSpectreAnalysis - GadgetCandidateNS

4.5.1 DefaultCacheEncodingAnalysis

This Analysis' goal is to identify 14 out of the 15 gadgets by Paul Kocher[41] as well as two extra that are defined in the FastSpec [19] paper. These gadgets are all derivations of the v01 gadget that can be seen in listing 4.7. For this gadget and its variants, the attacker controls `x` which is used to mistrain the branch predictor and eventually trigger an out-of-bounds memory access (`array1[x]`) during speculative execution. This access' result is then used to index a second array that encodes the memory contents accessed in the previous memory access into the cache. These contents can then be leaked by an attacker using a cache-based timing attack like Flush+Reload. The key aspect of this gadget, besides the attacker-controlled index, is the dependency between the second memory access and the first. This pattern can be seen in all of the previously mentioned gadgets, except in the v10 gadget shown in listing 4.8 in fig. 4.6, which does not leak data by encoding the leaked data in the cache directly. Instead, the v10-style gadget variants leak data by accessing or not a specific memory address depending on the result of a comparison. This comparison is performed between the value of the data to be leaked and an attacker-controlled value. This gadget variant will be explored in Section 4.5.2.

```
1  void victim_function_v01(size_t x) {
2    if (x < array1_size) {
3      temp &= array2[array1[x] * 512];
4    }
5  }
6
7  ASM:
8  mov    eax, DWORD PTR array1_size
9  cmp    rcx, rax
10 jae    SHORT $LN2@victim_fun
11 lfence
12 lea    rdx, OFFSET FLAT:__ImageBase
13 movzx eax, BYTE PTR array1[rdx+rcx]
14 shl    rax, 9
15 movzx eax, BYTE PTR array2[rax+rdx]
16 and    BYTE PTR temp, al
17 $LN2@victim_fun:
18 ret    0
```

Listing 4.7: v01 gadget

```
1  void victim_function_v10(size_t x,
     uint8_t k) {
2    if (x < array1_size) {
3      if (array1[x] == k)
4        temp &= array2[0];
5    }
6  }
7  ASM:
8  mov    eax, DWORD PTR array1_size
9  cmp    rcx, rax
10 jae    SHORT $LN3@victim_fun
11 lea    rax, OFFSET FLAT:array1
12 cmp    BYTE PTR [rcx+rax], dl
13 jne    SHORT $LN3@victim_fun
14 movzx eax, BYTE PTR array2
15 and    BYTE PTR temp, al
16 $LN3@victim_fun:
17 ret    0
```

Listing 4.8: v10 gadget

Figure 4.6: Side-by-side of the v01 gadget from Kocher [41], introduced in the Spectre [3] paper, and the v10 gadget variant from Kocher [41]
.

The DefaultCacheEncodingAnalysis works by tracking dependencies between tainted memory accesses, i.e., memory accesses where the address being accessed is potentially attacker-controlled. First, we must define the concept of dependency. A variable is deemed dependent on another variable or expression if its value is affected by it. Take the two Vex statements in listing 4.9. Line 1 shows the t24 variable's value being assigned to the contents of the memory at the address given by the value of the t18 variable, so t24 depends on that memory load expression. In line 2, the variable t25 is assigned the result of the arithmetic operation `t24 + 1`, which means its value is affected by the value of t24, which in turn depends on the memory load performed in line 1. Therefore, t25 is also dependent on the memory

load `LDle:I64(t18)`.

```
1  t24 = LDle:I64(t18)
2  t25 = Add64(0x1,t24)
```

Listing 4.9: Two Vex statements. A Load statement writes to variable t24. The value of the t24 variable is then used to calculate the value of the variable t25.

The DefaultCacheEncodingAnalysis has an internal structure that keeps track of all the variables whose value is affected by a tainted memory access, i.e., dependent on such memory access. When visiting a memory access statement on a tainted address, an entry is created in this structure, mapping the temporary variable corresponding to the `lhs` of the `Load` statement with a unique identifier. Each of these identifiers is also associated with a taint source.

Assuming v_0 is tainted by the register rdi and that no memory accesses have been visited so far, when visiting the statement $v_1 = LD(v_0)$, the structure would be updated with $\{v_0 : n_0, n_0 : rdi\}$. Subsequent statements that depend on v_0 will modify this structure, keeping it up to date. If the next visited statement is $v_1 = v_0$, the structure would be updated to $\{v_1 : v_0, v_0 : n_0, n_0 : rdi\}$. Having this structure, we can use the algorithm in listing 4.10 to query the structure for a given variable's dependencies.

If a variable is modified to no longer depend on a memory access, the structure is updated to reflect that. For example, if later on in the execution we visit the statement $v_1 = Get(rax)$, then v_1 will no longer depend on the memory access n_0 tainted by rdi, and the structure is updated to look like this: $\{v_1 : "I", v_0 : n_0, n_0 : rdi\}$. A call to the lookback method on this last structure will stop when it reaches the "I" entry, not reaching a memory access identifier, so the variable is deemed independent of any memory loads.

```python
def _lookback(dependencies, tmp, control):
    ''' returns the memory access ID from which tmp depends '''
    if tmp not in dependencies:
        return -1, None

    max_steps = len(dependencies) + 1
    t = dependencies[tmp]
    prev = tmp
    steps = 1

    # IDs are of the form "n#" where # is an integer
    while "n" not in t and t != control:
        prev = t
        if steps >= max_steps:
            raise Exception("MAX STEPS " + str(tmp))
        steps += 1
        if t in dependencies:
            t = dependencies[t]
        else:
            # not found
            return -1, None

    # found a loop, so return
    if t == control:
        return -1, t

    return t, prev
```

Listing 4.10: Lookback method for querying the dependency structure

For all statement types except `WrTmp`, the DefaultCacheEncodingAnalysis calls the auxiliary Analysis on the statement being visited.

When visiting a `WrTmp`, the DefaultCacheEncodingAnalysis starts by calling the auxiliary Analysis `WrTmp` visitor, which will return whether we are in the presence of a memory access and if it has a dependency or not. The DefaultCacheEncodingAnalysis then uses this information to build a directed graph, where each node represents a memory access, and each edge represents a dependency relation.

At the end of the `trace` 's execution, the DefaultCacheEncodingAnalysis generates a report containing all potential gadgets it has identified, which correspond to all of the possible paths in the dependency graph with at least two nodes.

Example

As an example of the DefaultCacheEncodingAnalysis's behavior, we will follow the execution of a trace from a binary containing the gadget in listing 4.7. We are looking specifically at a trace composed of the `IRSBs` in fig. 4.5. This trace will be referred to as `trace A`.

The `Executor` and the `DefaultTaintPolicy` will run their course, building a memory model, propagating taint and calling the `DefaultCacheEncodingAnalysis`'s visitors. When the execution reaches the statement at line 5 of the block at address 0x40123e (`t24 = LDle:I8(t18)`) the DefaultCacheEncodingAnalysis will query the `Executor` 's taint policy for the taint value of `t18`, which is currently tainted by `rdi`. An entry is then created in the dependency structure as well as a node in the dependency graph, representing this memory access. Let's call this node `LD0`.

Execution will resume, but now the DefaultCacheEncodingAnalysis will also update its internal dependency structure, keeping track of all the variables that depend on `t24`. This is the case for the variable `t42` which is used in line 45 as the address of another memory access (`t48 = LDle:I8(t42)`). This relation implies that the memory access in line 45 depends on the memory access in like 28. To represent this relation, a new node is created in the dependency graph (referred to as `LD1`), and a directed edge is created, connecting `LD0` to `LD1`.

When the trace finishes executing, a depth-first search will be performed on the graph, and the path from `LD0` to `LD1` will be found and flagged as a potential Spectre gadget by being added to a GadgetCandidate report.

4.5.2 ComparisonResultAnalysis

As shown in listing 4.8 (fig. 4.6), gadget v10, requires two attacker-controlled variables instead of one: `x` and `k`. This gadget works by leaking the result of the comparison `array1[x] == k` during speculative execution by accessing a constant memory address based on the result of this comparison. The attacker can then, like in the case for the previously mentioned gadget variants, perform a cache-based timing attack like Flush+Reload to leak the result of the comparison, and consequently `array1[x]`. Here, there is no dependency between memory accesses, so the `DefaultCacheEncodingAnalysis` cannot identify this gadget variant.

SpecTacle's approach to detecting this gadget type is to flag every memory access that is in the scope of a `tainted guard`. A tainted guard corresponds to the guard of a conditional branch where a comparison involving at least one tainted operand is performed. Subsequent memory accesses performed in the context of this branch will inevitably leak data.
In the example in listing 4.8, the tainted guard corresponds to line 3, where the memory access respon-

sible for the data leak is present in line 4.

It is crucial that we only flag tainted guards that can be triggered in the context of speculative execution. Since every SpecTacle search starts from a conditional branch, it will encounter at least one comparison before the branch itself. Because the DefaultTaintPolicy assumes every register is tainted, these comparisons could potentially be flagged as tainted guards, which would be an error and increase the number of false positives. The ComparisonResultAnalysis analysis solves this issue by tracking whether the exploration has entered a zone where speculative execution can occur and only then starts looking for tainted guards.

Whenever a tainted guard is found, the ComparisonResultAnalysis also stores the `abort address`. The abort address corresponds to the address from which statements are no longer affected by the tainted guard. So, for example, given the control flow in fig. 4.7, when encountering the tainted guard that can lead execution to block B, the address of block C would be stored. The execution would then be halted when reaching block C.

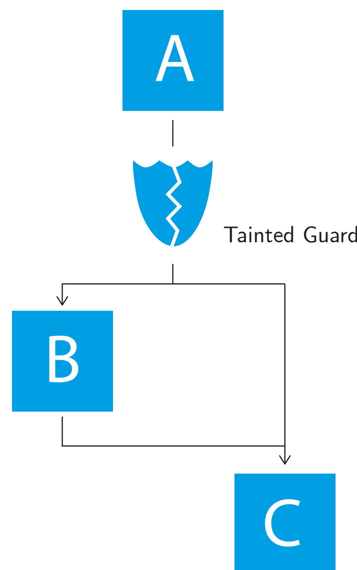

Figure 4.7: Example of a possible v10 gadget's control-flow graph

ComparisonResultAnalysis behavior can be interpreted as a state machine. This state machine has three states that are illustrated in fig. 4.8. Its states behavior can be described as follows:

1. **State 1** visits statements until a conditional branch is found, where it transitions to state 2.

2. **State 2** continues visiting statements until all the requirements are met for a tainted guard to be identified. When a tainted guard is detected, the ComparisonResultAnalysis transitions to state 3.

3. **State 3** continues visiting statements until either a memory load statement or the abort address are reached. A v10 gadget is then detected or not, respectively.

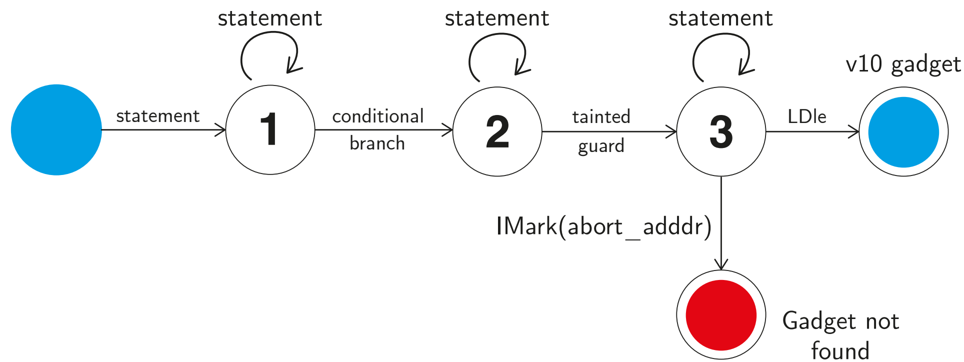

Figure 4.8: High-level ComparisonResultAnalysis analysis state machine. **1)** In state 1 ComparisonResultAnalysis is waiting for a conditional branch. **2)** While in state 2, ComparisonResultAnalysis is looking for a tainted guard. **3)** While in state 3, if a LDle statement is visited, a potential v10 gadget is flagged.

```
1 if (x < length)
2     if(array[x] > y)
3         flag &= true
```

Figure 4.9: NetSpectre leak gadget that uses the cache as a covert channel [46].

4.5.3 NetSpectreAnalysis

The purpose of the NetSpectreAnalysis is to identify NetSpectre [25] leak gadgets. NetSpectre [25] leak gadgets have a very similar structure to Spectre V10-type gadgets. In essence, they leak data by modifying microarchitectural state depending on the result of a comparison performed with a tainted guard [46].

Take the code snippet in fig. 4.9 from [46]. If the attacker controls both `x` and `y` the branch predictor can be mistrained to perform an out-of-bounds memory access through `array[x]`, and based on the result of the comparison in line 2, `flag` will either be brought to the cache or not, which will cause a difference in the runtime of future operations that use the `flag` variable. An attacker can measure these runtime differences to obtain information about the contents of `array[x]`.

Schwarz et al. also proposes an alternate side-channel based on the execution time of AVX2 instructions [42]. This channel uses the fact that the top half of the AVX2 unit becomes inactive after not being utilized for a specific duration [44], so AVX2 instructions that use this unit take longer if the unit is inactive [45]. This timing difference can be exploited like previously to leak the results of a comparison [25]. An example of a leak gadget that uses this side-channel can be seen in fig. 2.4 [25].

NetSpectreAnalysis uses a tainted guard mechanism like the ComparisonResultAnalysis, but instead of looking for memory accesses to encode data in the side channel, it looks for statements that change microarchitectural state.

SpecTacle provides support for both the cache-based and the AVX2 side channels.
As mentioned previously, the cache-based side-channel works by bringing data to cache. This can be

done with both memory loads and stores. ComparisonResultAnalysis already implements the memory load case, so NetSpectreAnalysis only implements the memory store case: whenever a memory store is detected within the scope of a tainted guard, a NetSpectre leak gadget is flagged.

Support for the AVX2 side channel is implemented similarly to the cache-based side channel. Instead of flagging a gadget on a Store statement, SpecTacle queries the `Intel Intrinsics Guide` [33] for AVX2 instructions. When an AVX2 instruction is found, a NetSpectre gadget is found and identified with the tag `(AVX)`.

4.6 Validator

The `Validator` is the last stage in SpecTacle's workflow. The potential gadgets identified by the analyses are given to the Validator, whose task is to validate the feasibility of the gadgets' traces in the context of speculative execution, i.e., if it is possible to follow the path specified by a given trace given the path constraints imposed by conditional branch.

When facing a conditional branch, execution can follow in four different paths: two regular paths and two transient paths:

- The `regular` paths correspond to the two possible addresses execution can continue to, architecturally: the `then` branch and the `else` branch. These correspond to the red and green paths in fig. 4.10. We are not interested in these paths since a focal characteristic of a Spectre gadget is that it is executed in a transient path.

- The `transient` paths represent executions where a branch misprediction occurred. This will cause the `then` and `else` branches to be executed in the wrong context. Using fig. 4.10 as reference, these would correspond to the blue paths, where the `then` branch is executed with the constraint `c = false`, and the else path being executed with the constraint `c = true`.

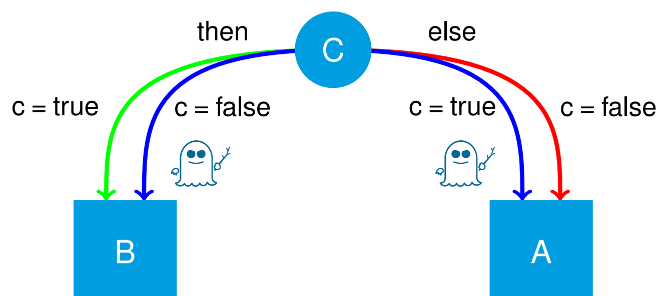

Figure 4.10: Possible execution paths following a conditional branch C.

To validate the feasibility of these transient paths, the Validator makes use of `angr's` symbolic execution engine. Fig. 4.11 illustrates validation process for a transient path represented by the trace [B1, B2, B3]. The several steps the Validator takes to evaluate this path are explained in detail:

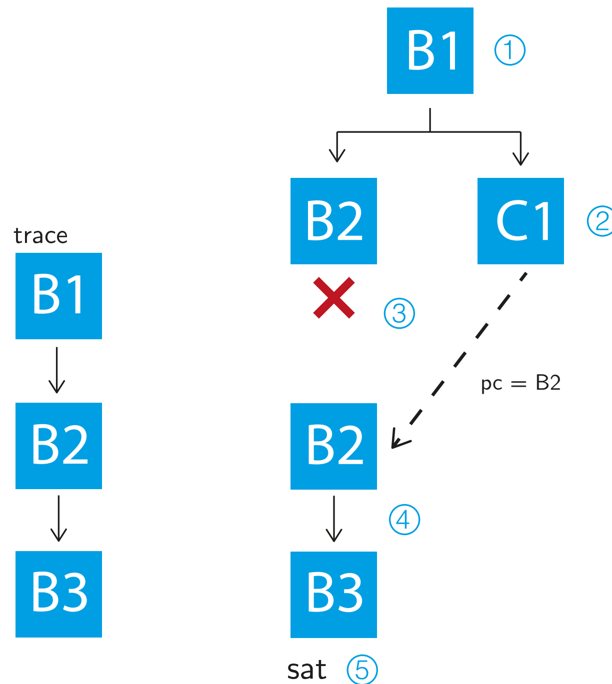

Figure 4.11: Scheme of how the Validator simulates speculative execution and validates the feasibility of a given transient path.

1. Execution starts at the first basic block of the given trace (B1). The gadget's conditional branch must belong to this basic block.

2. The execution proceeds until the conditional branch and executes the branching instruction. From this, two different states should be created: one following the `then` branch (B2) and another following the `else` branch (C1), both with the respective constraints that caused execution to flow to the respective path. One of these two states must correspond to the second address of the trace. In this example, this address is B2.

3. The Validator then identifies the state that continues to the correct address after the conditional branch (B2), drops it, and selects the alternative state to continue execution by modifying its program counter register to point to the address of the B2 basic block. This modification forces the rest of the path to be executed with the constraints imposed by the `branch not taken` path while following the `branch taken` path, simulating a transient path. Switching the path constraints is equivalent to executing one of the blue paths in fig. 4.10.

4. From this state, the Validator now asks the simulation manager to find a path that follows the gadget's trace and executes its dependent memory loads or tainted guards in the correct order.

5. At the end of the symbolic execution, if the path constraints are satisfiable, the Validator deems this path feasible and approves this gadget candidate. Otherwise, this gadget candidate is discarded.

4.7 Optimizations

One of the main objectives of this work was to develop a tool that would be faster than the other available tools. In this chapter, we will discuss optimizations implemented in SpecTacle to increase performance and reducing runtime.

4.7.1 Memory Load Limit

The number of micro-ops that can be executed speculatively in modern CPUs can reach the hundreds [31], but since there is not a one-to-one relationship between micro-ops and instructions, it is impossible to define a precise limit for the maximum number of instructions on a transient path.

Memory load instructions take orders of magnitude longer to resolve than accessing a CPU register [89, 90]. Therefore, a limited number of memory loads can be performed in a transient path. For example, Ragab et al. [91] managed to fit 12 load instructions in a transient window by manipulating the FPU through a chain of dependent floating-point operations. This number seems to be a reasonable upper limit on the number of loads possible on a transient path in modern CPUs, so SpecTacle adds an extra Analysis to the Executor that causes the execution of a trace to be halted when the thirteenth memory load is visited in that trace.

4.7.2 Parallelization

Each trace can be processed independently by the Executor. Therefore, to decrease execution time SpecTacle allows dividing the workload between a configurable number of threads. The results obtained by parallelizing the trace processing were significant and will be discussed in chapter 5. Although SpecTacle supports running the Executor and analyses in parallel, the Explorer and the Validator remain as single-threaded modules. Providing multi-thread support for both modules can experience great performance benefits from running in parallel. This support is left as future work.

4.7.3 State Memoization

Due to how the Explorer generates the traces, they will inevitably share basic blocks. An optimization that could significantly reduce runtime in expense for memory usage would be to cache the Executor, taint policy, and analyses states of IRSBs that will be repeatedly visited.

Ultimately, our implementation of state memoization was faulty, which affected both the runtime and memory usage performance of SpecTacle for the worse. For this reason, a working implementation of state memoization is not provided in SpecTacle and is left as future work.

Chapter 5

Results

In this chapter we discuss SpecTacle's performance and scalability, as well as comparing gadget detection results with FastSpec. We will discuss the testing data, as well as the experimental setup. Finally, we provide examples of Spectre-PHT gadget detection in a Java program, as well as validating our NetSpectre [25] AVX-based gadget detection analysis. We end this chapter by looking at a real world example of a gadget detected by SpecTacle in Brotli [92].

5.1 Problem Description

Automated Spectre-PHT gadget detection is still a novel research topic. Much research has been developed in this field in the last two years, and many approaches have been explored. Tools like SpecFuzz and the more recent SpecTaint [16, 23] explore dynamic approaches to gadget detection, but they suffer from problems inherent to dynamic analysis. They are bonded by test case quality [23] which can limit code coverage, and gadgets present in unexplored code are not found. SpecTaint [23] claims that the dynamic approach can be combined with static approaches to reduce the amount of uncovered code, resulting in fewer false negatives. As for static analysis tools, oo7 [20] performs taint analysis statically at the binary level to identify Spectre-PHT gadgets. However, oo7 is reported to have high execution times and introduce both false negatives, and false positives [19, 23]. Another approach explored in recent research is symbolic execution. SPECTECTOR [17] introduced the concept of Speculative Non-interference (SNI) to model Spectre gadgets using symbolic execution, while Pitchfork [70] introduces *speculative consant-time* with the same purpose to detect Spectre gadgets in speculative execution. These approach suffer from the inherent flaw of symbolic execution that is path explosion. Also, SPECTECTOR suffers from having poor instruction coverage [17, 18], i.e., many instructions are not supported, which makes this tool not appropriate for real-world use. KLEESpectre introduces an explicit approach to model speculative execution in a symbolic execution context that heightens the problem of path explosion. Haunted RelSE [18] improves upon KLEESpectre's explicit model by developing a novel exploration technique to attenuate the problem of path explosion, which improved precision, recall, and execution times [18]. FastSpec [68] takes a novel approach to Spectre gadget detection by applying

NLP techniques to Spectre gadget detection. FastSpec trains a classifier based on BERT [66] with over one million generated gadgets to classify Spectre gadgets. This approach is promising because it keeps execution times low. However, it introduces false negatives when considering classifier outputs with high confidence levels. To reduce the number of false negatives, we can look at lower confidence levels which inevitably increases the number of false positives. Also, in Section 3.3 we argue that FastSpec's training set introduces avoidable false positives.

Our main goal with SpecTacle is to provide a static analysis alternative based on taint analysis that significantly reduces the number of false negatives compared to other static analysis tools while keeping runtimes low. Secondly, our tool provides context about the flagged gadgets, like execution path and, when possible, what data must be attacker-controlled. This information facilitates the combination of SpecTacle with dynamic analysis or symbolic execution tools for a more precise analysis. Also, the problem of gadget detection for JITed [59] languages has not been tackled yet, so SpecTacle proposes a technique to provide support for such languages by developing a proof-of-concept that detects a Spectre-PHT in a sample Java class. This technique is applicable to other JITed programming languages. Another problem tackled by SpecTacle is the detection of NetSpectre [25] gadgets, in particular gadgets that use the AVX-based side-channel [25]. Finally, at the cost of execution time, SpecTacle provides a symbolic path validation mechanism to reduce the number of false positives further.

With SpecTacle we successfully achieved our goals by providing a tool that produces less false negatives than the state-of-the-art tool FastSpec [19] that is reported to have very high recall values while being faster than other state-of-the-art tools. In addition, SpecTacle manages high recall values while keeping low execution times for similar speculation window sizes.

In our experiments, we could not successfully run several of the state-of-the art tools. Our SpecFuzz [16] setup could not achieve the code-coverage results reported by Oleksenko et al. [16] and [23]. SpecTaint [23] is still not open-sourced and we did not receive a response from the authors. Oo7 [20] is one of the tools that SpecTacle was meant to improve upon, but we could not successfully setup this tool following the instructions from the official GitHub repository ([93]) due to outdated dependencies. However, we managed to run a community-modified version ([94]) but could not replicate the results from the original paper ([20]) using this version, failing to detect vulnerabilities in the 15 gadgets from Kocher [41], so we deemed it not suitable for comparison. Finally, Binsec/Haunted only supports 32-bit binaries, which SpecTacle also supports but it is not optimized for, so we decided a fair comparison could also not be performed with this tool.

5.2 Technologies

Python [95] 3.9.7 was the language of choice for the implementation of SpecTacle due to the number of supported modules and frameworks.
The angr[77] Framework was used to lift the input binary's basic blocks to `PyVex`[81] and for its symbolic execution engine. The version used was 9.0.9506.

For disassembly and analysis of the input binary's properties, the `r2pipe`[96] package was used to script the radare2[97] debugger from Python. Version 1.6.2 of r2pipe was used to interact with radare2 version 5.4.3.

5.3 Experimental Setup

Experiments were performed on a desktop computer equipped with an AMD Ryzen 3700X 8/16, 32GB of RAM, and an Nvidia GeForce GTX 1650 GPU. The speculation window was variable, according to the tables in the following sections.

We perform evaluations on the *Real-world V1 Dataset* from SpecTaint [23] and SpecFuzz [16] which are comprised of six libraries: OpenSSL [98] v3.0.0, a Brotli [92] v1.0.7, JSMN [99] v1.1.0, HTTP [100] v2.9.2, libHTP [101] v0.5.30, and libYAML [102] v0.2.2. Finally, for comparing our results with FastSpec's [19] we used some benchmarks of the Phoronix Test Suite v10.2.0 [103].

Even though SpecTacle supports all formats that angr [82] can load with CLE [82], we focused our development and testing on the x86-64 architecture and ELF binaries.

5.4 Performance and Scalability

One of our goals was to make a scalable tool that could be used for big code bases, meaning that execution times and resource requirements do not exponentially scale with the size of the codebase. Tables 5.1, 5.2 and 5.3 illustrate how SpecTacle scales according to the selected performance metrics. The defined metrics are:

- The program size in bytes;
- Execution times in seconds;
- The number of conditional branches (**#B**);
- The number of paths explored (**#P**);
- The number of functions flagged as containing a gadget (**#F**);
- The number of gadgets flagged by the DefaultCacheEncodingAnalysis (**#D**);
- the number of gadgets flagged by the ComparisonResultAnalysis (**#C**).

5.4.1 Explorer

As illustrated in table 5.1, table 5.2 and table 5.3 we conclude that SpecTacle's performance grows exponentially with the number of conditional branches present in the given binary. Also, the size of the

speculation window has a significant impact on performance because it leads to path explosion. By design, the number of conditional branches in the input binary will always be the main performance deterioration factor for SpecTacle. However, it is possible to reduce the number of explored paths by identifying the nature of a conditional branch before exploring it. At this point in development, SpecTacle does not check the nature of the conditional branch, so branches that do not perform bounds checks are analyzed, significantly decreasing performance and precision. This analysis is left as future work as part of developing a more robust and efficient Explorer module.

Library	Size(byte)	Time(sec)	#B	#P	#F	#D	#C
libHTP	335K	1.97	149	160	1	19	1
HTTP	7.3K	1.16	20	28	1	5	0
JSMN	26K	2.99	101	1939	1	8	0
Brotli	4,3M	78.71	6450	49278	295	3186	62
YAML	162K	22.09	2333	26067	101	1145	7
OpenSSL	162K	5.33	404	2379	19	81	6

Table 5.1: SpecTacle results for a speculation window of 15 instructions using the DefaultTaintPolicy, without running the Validator.

Library	Size(byte)	Time(sec)	#B	#P	#F	#D	#C
libHTP	335K	2.22	149	522	1	19	1
HTTP	7.3K	1.17	20	39	1	6	0
JSMN	26K	45.84	101	68887	1	72	2
Brotli	4,3M	722.17	6454	679611	305	4696	290
YAML	162K	1839.55	2333	2593070	102	1351	128
OpenSSL	162K	11.67	404	10010	31	152	17

Table 5.2: SpecTacle results for a speculation window of 35 instructions using the DefaultTaintPolicy, without running the Validator.

Library	Size(byte)	Time(sec)	#B	#P	#F	#D	#C
libHTP	335K	2.149	149	522	1	7	1
HTTP	7.3K	1.157	20	39	1	4	0
JSMN	26K	45.470	101	68887	1	71	2
Brotli	4,3M	669.40	6454	679611	284	2502	8
YAML	162K	1793.29	2333	2593070	93	834	8
OpenSSL	162K	11.104	404	10010	29	57	0

Table 5.3: SpecTacle results for a speculation window of 35 instructions using the StrictTaintPolicy, without running the Validator.

5.4.2 Validator

The goal of the Validator is to prune detected gadget traces that are not feasible during speculative execution, i.e., given the constraints of the conditional branch, the given path identified by the previous analysis cannot be executed. This module adds precision at the cost of execution time. Table 5.4 illustrates the tradeoff between the number of flagged gadgets and the execution time. Unfortunatelly, we could not obtain the values for the Brotli binary due to a floating point exception launched from angr's [82] internals. We could not solve this issue.

The Validator is responsible for the bulk of SpecTacle's execution time. These results show that this approach is not optimal for precision in Spectre gadget verification and negatively affects the tool's scalability. The current Validator does not provide enough precision increase to justify the performance loss it entails. This performance loss is primarily due to the overhead imposed by setting up a new simulation manager [82], and symbolic state [82] for every analyzed path. Since we are paying this overhead, a more in-depth analysis such as SNI [17, 18] or speculative constant-time [70] semantics checking can be performed, and optimizations such as Haunted RelSE [18] can be used for a more precise and efficient gadget validation module. Another reason for the Validator's high execution time, compared to the Executor and the analyses, is that we could not implement a fully working parallel version of the Validator due to problems with exception handling. If we can validate gadget traces in parallel, performance will be greatly increased. This optimization is left as future work. Nevertheless, for smaller speculation window sizes or smaller binaries, making use of the Validator to increase precision is shown in Section 5.5 to be effective.

Binary	15 instruction window				35 instruction window			
	Time(sec)	#F	#D	#C	Time(sec)	#F	#D	#C
libHTP	69.97	1	19	1	117.45	1	19	1
HTTP	9.17	1	4	0	1.4552	1	4	0
JSMN	7.43	1	71	0	260.30	1	64	1
Brotli	1634.13	248	2502	0	*	*	*	*
YAML	4360.06	82	834	0	19937.77	102	1266	82
OpenSSL	10.29	2	8	0	1106.18	20	109	12

Table 5.4: SpecTacle results for a speculation windows of 15 and 35 instructions using the DefaultTaintPolicy and running the Validator. Entries marked * represent values that we could not obtain.

5.5 Gadget Identification

In this subsection, we provide an overview of SpecTacle's precision and recall values, as well as how it compares to other state-of-the-art tools. Precision is calculated as $TP/(TP + FP)$, and recall as $TP/(TP + FN)$ with TP, FP and FN meaning true positive, false positive, and false negative, respectively. The lack of an annotated testing dataset proved to be a real challenge to the accurate evaluation of SpecTacle.

FastSpec [19] uses as ground truth the conditional branches flagged as vulnerable by SpecFuzz [16] on some of the benchmarks programs of the Phoronix Test Suite [103], i.e., gadgets flagged by SpecFuzz are deemed as true positives, and all other gadgets detected are deemed false positives. FastSpec then calculates precision and recall values for the given binaries according to this ground truth. The main flaw with this approach is that SpecFuzz, due to its flawed gadget modeling, is reported to be prone to both false positives and false negatives [16, 23].

5.5.1 Dataset

In the SpecTaint paper [23], Qi et al. [23] propose two different approaches: manually verifying identified gadgets and inserting gadgets manually. Both of these approaches have disadvantages. The former is prone to human error and is not scalable for large codebases. However, the latter can introduce biases. SpecTaint [23], Qi et al. [23] uses a vulnerability injection approach based on LAVA [104] that guarantees that injected gadgets can be reached by SpecTaint and are attacker-controlled. However, for tools like SpecTacle and FastSpec [19] which do not directly test for attacker-controlled data and analyze the entire binary, Spectre gadget injection can be performed with a more relaxed approach.

Our goal with SpecTacle is to provide a static analysis tool with high recall rates without high costs to precision compared to other static analysis tools with similar execution times. Table 5.5 illustrates the scale of the number of gadget candidates flagged by FastSpec with the mentioned threshold of 0.48 [19] and SpecTacle. These values are too high for manual validation, so we chose five Phoronix Test Suite benchmark programs to which we injected a total of 10 Spectre-PHT gadgets to enable precision and recall calculation. Alongside these five programs, a litmus test program is also added containing all of the 15 Kocher [41] Spectre-PHT gadgets and alongside two gadget variants introduced by FastSpec [19] and a modified version of the v01 gadget from Kocher [41] with floating-point operations, for a total of 6 functions with Spectre gadgets. Listing 5.1 shows an example of a gadget injected in the *mbw* benchmark from the Phoronix Test Suite v10.2.0 [103] (lines 4–7 and 17).

```
void __attribute__ ((noinline)) leakByteNoinlineFunction(uint8_t k)
{ temp &= arrayPu[(k)* 512]; }
void victim_function_v03(size_t x) {
     if (x < arrayPr_size)
          leakByteNoinlineFunction(arrayPr[x]);
}
...
int main(int argc, char **argv){
...

while((o=getopt(argc, argv, "haqn:t:b:")) != EOF) {
    switch(o) {
    ...
    case 'n': /* no. loops */
        nr_loops=strtoul(optarg, (char **)NULL, 10);
        victim_function_v03(nr_loops);
        break;
    ...
```

Figure 5.1: Spectre-PHT gadget injected from Kocher [41] in the Phoronix Test Suite mbw benchmark [103].

Binary	FastSpec		SpecTacle		SpecTacle with Validator	
	#F	Time(sec)	#F	Time(sec)	#F	Time(sec)
libHTP	6	32.50	1	2.4	1	30.2699
HTTP	1	8.08	1	1.45	1	10.12
JSMN	4	11.86	1	55.24	1	285.90
Brotli	165	3450.66	284	788.23	*	*
YAML	34	567.66	93	2307.86	93	15701.86
OpenSSL	3	156.93	29	12.2	4	173.06

Table 5.5: Table comparing the execution time and number of functions with flagged gadgets for FastSpec with a confidence level of 0.48, and SpecTacle using the StrictTaintPolicy for a speculation window of 35 instructions, with and without the Validator. Entries marked * represent values that we could not obtain.

5.5.2 Evaluation

Since FastSpec [19] assigns a confidence level to a window of tokens, it is not trivial to obtain a direct translation between a high confidence level and a code location. Therefore we take the first token in each window and assign it its corresponding function from the binary. Functions that contain a token window with a confidence level above a given threshold are considered vulnerable. We then compare the functions deemed vulnerable by FastSpec with the functions where SpecTacle identified gadgets. We compared SpecTacle against three FastSpec confidence level thresholds: 0.48 [19], 0.6 and 0.8. Tol et al. [19] claimed that, in their experiments, all gadgets were detected with a confidence level of 0.48 or less, so we choose this value as the lower bound for comparing our results with FastSpec's. Additionally, we used token windows of size 80 and a shift value of 1 for the sliding window, also as recommended by Tol et al. [19]. To recover function information, we modified the scripts in FastSpec's GitHub repository [68] to allow for this recovery without altering the confidence levels given by the original

scripts. Finally, we run SpecTacle with a speculation window size of 35 instructions for a fair comparison with FastSpec, since FastSpec assigns a confidence level to a window of 80 tokens. Most instructions are comprised of 2 or 3 tokens, so we find a 35 instruction window an appropriate value for comparison with FastSpec's 80 token windows.

The precision and recall values for the dataset containing the injected gadgets are shown in table 5.7. We noticed that FastSpec consistently assigns confidence levels above 0.7 to non-program-specific functions like *__libc_csu_init* and *__libc_csu_fini*, which inflates the number of false positives, decreasing precision.

Table 5.7 shows that SpecTacle manages higher recall values than FastSpec using the DefaultTaintPolicy, identifying all of the injected gadgets.

Tables 5.6 and 5.5 show the execution times for SpecTacle and FastSpec for the given binaries. In general, SpecTacle is faster for smaller binaries, with and without running the Validator. For larger binaries with more density of conditional branches SpecTacle is slower, especially if we run the Validator module. Unfortunatelly, for the *XSBench* binary, the Validator crashes with a floating point exception originated in angr's [82] internals. We could not find a solution to this issue.

To test the viability of the StrictTaintPolicy, we run SpecTacle with this policy for the same binaries and speculation windows. The results can be seen in table 5.8. We can conclude that the StrictTaintPolicy is a viable alternative to the DefaultTaintPolicy if the goal is to reduce the number of false positives at the cost of a possible reduction in the recall values.

Binary	FastSpec	SpecTacle DTP	SpecTacle DTP Validator
	Time(sec)	Time(sec)	Time(sec)
cachebench	33.48	2.53	5.97
mbw	13.75	1.79	2.14
postmark	38.78	2.96	24.11
t-test	15.58	1.65	2.36
ltimus	11.98	1.45	2.79
XSBench	41.75	5.26	*

Table 5.6: Execution times of FastSpec[19] and SpecTacle for selected Phoronix Test Suite binaries. SpecTacle was run with a speculation window size of 35 instructions and the DefaultTaintPolicy (DTP). * means that the specific value could not be obtained

5.6 Case Studies and Extensions

In this section, we discuss our proof of concept for Java Spectre-PHT gadget detection and discuss how it can be applied to other JIT-compiled languages. After, we look into NetSpectre and AVX-based gadgets. Finally, we look at a real-world gadget example from Brotli [92].

Binary	FastSpec 0.48		FastSpec 0.6		FastSpec 0.8		SpecTacle	
	P	R	P	R	P	R	P	R
cachebench	0.095	0.6(6)	0	0	0	0	0.50	1.00
mbw	0.20	0.50	0	0	0	0	0.666	1.00
postmark	0.91	0.50	0.2	0.5	0	0	0.182	1.00
t-test	0.1(1)	1.00	0	0	0	0	0.333	1.00
litmus	0.037	0.5	0.06	0.5	0	0	1.00	1.00
XSBench	0.609	0.7(7)	0.5	0.4(4)	0.6	0.16	0.154	1.00

Table 5.7: Comparison between FastSpec's precision and recall values for confidence levels of 0.48, 0.6 and 0.8, with SpecTacle's using the DefaultTaintPolicy for a speculation window of 35 instructions and without the Validator. **P** and **R** represent precision and recall respectively.

Binary	SpecTacle DefaultTP			SpecTacle StrictTP			SpecTacle DefaultTP (V)		
	P	R	Time(sec)	P	R	Time(sec)	P	R	Time(sec)
cachebench	0.50	1.00	2.53	0.75	1.00	2.53	0.75	1.00	5.97
mbw	0.666	1.00	1.79	0.66	1.00	1.75	0.66	1.00	2.14
postmark	0.182	1.00	2.96	0.1429	0.50	2.97	0.286	1.00	24.11
t-test	0.33	1.00	1.65	0.33	1.00	1.68	0.33	1.00	2.36
litmus	1.00	1.00	1.45	1.00	1.00	1.40	1.00	1.00	2.79
XSBench	0.154	1.00	5.26	0.286	1.00	5.26	*	*	*

Table 5.8: Comparison of the precision and recall values, alongside execution time of the DefaultTaint-Policy and the StrictTaintPolicy without the Validator, and the DefaultTaintPolicy with the Validator (V) in the last column. Results obtained with a speculation window of 35 instructions. **P** and **R** represent precision and recall, respectively. * means that it was not possible to obtain a result.

5.6.1 Java Example

To test our Java Spectre-PHT gadget detection proof of concept, we used a simple Victim.java class shown in listing 5.1 where parts of the *main* function were removed for simplicity. Our goal is then to detect the gadget inside the *specFunction* that can be seen in lines 10 through 14. First, we need to compile this Java class. For this experiment, we used OpenJDK 11 [105]. We can compile our Java class using `javac`: `javac Victim.java`. Compilation outputs a `Victim.class` file, which will be used in the following stages of analysis.

Given a `.class` file we want to analyze, we can then proceed as follows:

1. Execute the `jparse.py` python script that implements Algorithm 1 described in Section 4.2.1. This script will extract the methods from the `.class` file, disassemble and process them, and finally add them to an assembly template that can be compiled to an executable. One executable is generated for each method in the `.class` file.

2. Given this executable, we can then give it to SpecTacle for analysis. Using the code example in listing 5.1, SpecTacle was able to detect the gadget in the *specFunction* method.

This approach is limited because it cannot explore paths that run through different methods not

located in the compiled binary. A better native code recovery method is left as future work, but this proof-of-concept demonstrates it is possible to detect Spectre gadgets in JITed Java code. Using this method, we argue that it is also possible to analyze other languages that offer runtimes with JIT-compilation, like C#, JavaScript running on the V8 engine, or any others that allow for recovery of the JIT-compiled code. Analysis of JIT-compiled code is especially relevant due to recent discoveries in Spectre exploitation of browsers through JavaScript as reported by Agarwal et al. [49] in Spook.js [49], especially given the increasing popularity of JIT-compiled languages like JavaScript.

```
1  import java.io.*;
2  import java.util.Random;
3
4  public class Victim {
5      static int len = 1;
6      static byte arrayPr[] = new byte[256+len];
7      static byte arrayPu[] = new byte[(256+1) * 4096];
8
9      static void specFunction(long x) {
10         if (x < arrayPu[256 * 4096]) {
11             tmp = tmp & arrayPu[arrayPr[((int)x)] * 4096];
12         }
13     }
14
15     public static void main(String[] args) throws Exception {
16         ...
17         while (true){
18             for(int offset= 0; offset < len; offset++){
19                     //calculate x
20                     ...
21                     specFunction(x);
22
23             }
24         }
25     }
26 }
```

Listing 5.1: Java Victim class used to test SpecTacle's Java gadget detection POC

5.6.2 AVX-based NetSpectre Example

For evaluating our NetSpectreAnalysis, in particular gadgets using the AVX-based side channel, we added the gadgets shown in fig. 5.2 to our litmus test file for verification. These gadgets were taken from the NetSpectre [46] paper. The gadget present in the *victim_function_netspectre_avx* function uses a VPAND 256-bit instruction [33] which is an AVX2 instruction and so utilizes the AVX2 unit, as required

```
1  void victim_function_netspectre(size_t x, uint8_t k){
2      if (x < arrayPr_size)
3          if(arrayPr[x] > k)
4              flag = 1;
5  }
6
7  void victim_function_netspectre_avx(size_t x, uint8_t k){
8      if (x < arrayPr_size)
9          if(arrayPr[x] > k)
10             b = _mm256_and_si256(a, b);
11 }
```

Figure 5.2: NetSpectre gadgets [46] added to the litmus test and used for SpecTacle's NetSpectre gadget detection proof-of-concept.

to encode data in the covert channel as explored in Section 2.5.3.

Running SpecTacle on the litmus test using the NetSpectreAnalysis correctly detects both gadgets in functions *victim_function_netspectre* and *victim_function_netspectre_avx*, labeling the gadget using the AVX2-based covert channel as such. To further test this analysis in real world applications, we analyzed the simdjson [106] library, a C++ library that leverages SIMD [42] instructions to parse JSON files, but no gadgets were detected.

5.6.3 Brotli

In their paper, Qi et al. [23] use SpecTaint [23] to detect Spectre gadgets in several open-source libraries, one of which being Google's Brotli [92]. Because of the lack of an annotated testing dataset, and therefore lack of ground truth, Qi et al. [23] manually checked the gadgets detected by SpecTaint [23]. As a result of this manual verification, the gadget shown fig. 5.3 was identified. The `transform_idx` variable is assumed to be tainted, and the bounds-check in line 13 can be mistrained to speculatively predict a `branch taken`, which leads to a call to the *BrotliTransformDictionaryWord* function (line 18) with an out-of-bounds `transform_idx` value. In *BrotliTransformDictionaryWord*, the *BROTLI_TRANSFORM_PREFIX* macro is used in line 26 to index the `prefix_suffix` array with the contents of `transforms[transform_idx * 3]` (lines 2, 4 and 5). Performing these memory accesses with an out-of-bounds value of `transform_idx` lead to out-of-bounds data relative to the `transforms` array being leaked, using the `prefix_suffix` array as a covert channel.

```
1
2  #define BROTLI_TRANSFORM_PREFIX_ID(T, I) ((T)->transforms[((I) * 3) + 0])
3
4  #define BROTLI_TRANSFORM_PREFIX(T, I) (&(T)->prefix_suffix[ \
5      (T)->prefix_suffix_map[BROTLI_TRANSFORM_PREFIX_ID(T, I)]])
6  ...
7  static BROTLI_INLINE BrotliDecoderErrorCode ProcessCommandsInternal(
8      int safe, BrotliDecoderState* s) {
9      ...
10   if (transform_idx < (int)transforms->num_transforms) {
11     const uint8_t* word = &words->data[offset];
12     int len = i;
13     if (transform_idx == transforms->cutOffTransforms[0]) {
14       memcpy(&s->ringbuffer[pos], word, (size_t)len);
15       BROTLI_LOG(("[ProcessCommandsInternal] dictionary word: [%.*s]\n",
16                   len, word));
17     } else {
18       len = BrotliTransformDictionaryWord(&s->ringbuffer[pos], word, len,
19           transforms, transform_idx);
20       ...
21   }
22   ...
23   int BrotliTransformDictionaryWord(uint8_t* dst, const uint8_t* word, int
       len,
24     const BrotliTransforms* transforms, int transform_idx) {
25     ...
26     const uint8_t* prefix = BROTLI_TRANSFORM_PREFIX(transforms,
         transform_idx);
27       ...
```

Figure 5.3: Spectre-PHT gadget found in Brotli by SpecTaint [92] [23]

To test the effectiveness of SpecTacle, we ran our tool with the Brotli binary to verify if this gadget would be flagged. SpecTacle detects the gadget in fig. 5.3 with a speculation window of 55 instructions and without the 12 memory load in speculation limit discussed in Section 4.7.1. This result was verified with both the DefaultTaintPolicy and the StrictTaintPolicy. FastSpec was not able to detect this gadget with a confidence level of 0.48 or higher.

Chapter 6

Conclusions

In this work, our main goal was to develop a static analysis tool based on taint analysis that would be an alternative to state-of-the-art tools by reducing the number of false negatives while keeping execution times comparatively low. We implemented SpecTacle, a tool that performs static taint analysis on VEX [76] instructions obtained by angr's [82] VEX lifter, PyVEX [85], which allows SpecTacle to support every executable format that can be loaded using CLE, angr's binary loader. Additionally, SpecTacle can forward the flagged gadgets to a validation module that checks the gadgets' feasibility using symbolic execution, reducing the number of false positives at the cost of execution time. Our tool can detect the 15 gadgets variants proposed by Kocher [41] as well as two other variants proposed by Tol et al. [19]. Variations of these gadgets are also detected.

Furthermore, we evaluate SpecTacle's efficacy by detecting a Spectre-PHT gadget reported by Qi et al. [23] in the real-world application Brotli [92]. These points are achieved while producing fewer false negatives than other state-of-the-art tools with comparatively low execution times. Besides, SpecTacle can provide detailed information about the detected gadgets like the address of the conditional branch that triggers speculation and possible memory addresses where the leaking and encoding memory load operations are performed. This additional information is meant to facilitate the integration with other tools or other analysis techniques that have proven to be more precise, like dynamic analysis or symbolic execution [18, 23]. To this effect, SpecTacle's static analysis can be used to reduce the search space for more computationally intensive analysis techniques while not discarding actual Spectre gadgets via false negatives. We have also discussed how these methods can be integrated directly in SpecTacle through its *Validator* module.

Additionally, to the best of our knowledge, no other state-of-the-art tool provides support for detecting NetSpectre AVX-based side-channel gadgets [25]. Therefore, we successfully provide a proof-of-concept analysis that is able to detect sample gadgets from Schwarz et al. [25].

Finally, to the best of our knowledge, only dynamic approaches have been applied to the detection of Spectre-PHT gadgets in JIT-compiled languages [21, 22]. In this work, we propose a static analysis approach to detect Spectre gadgets in JIT-compiled code and prove its efficacy with a proof-of-concept that successfully detects a Spectre-PHT gadget in a Java program.

Even though this work is focused on Spectre-PHT gadget detection, SpecTacle can be leveraged to perform arbitrary analyses on binaries using VEX and provides a flexible static taint analysis engine. This way SpecTacle is not only a Spectre gadget detection tool but a flexible program analysis tool.

6.1 Future Work

Two of the main drawbacks of our approach to Spectre gadget detection are path explosion and low precision. To combat these two issues, we could improve our explorer for Spectre-PHT by discarding conditional branches that do not perform bound-checks. In addition, we could provide a closer integration with angr [82] and leverage its control-flow-graph extraction tools to extract more accurate control flow graphs and obtain more information about the conditional branches before exploring them needlessly. Another approach to increasing precision would be to implement more accurate taint policies to avoid over-tainting.

Our Validator module does not provide significant value due to its high execution times. One way to decrease execution time would be to allow gadget validation to be performed in parallel. Also, a deeper analysis could be performed since we are already paying the overhead of symbolic execution through path constraint solving. For example, checking for Speculative-Non-Interference or speculative constant-time [17, 18] semantics, which are shown to be effective in detecting Spectre gadgets with high precision, so this analysis could be performed to validate gadgets flagged by SpecTacle's static analysis. Additionally, dynamic approaches that use hardware performance counters [21, 22] could be used to further validate flagged gadgets by ensuring that speculative execution can occur.

During testing, we noticed that a significant amount of redundant computation was performed. It is possible to decrease execution times by implementing state memoization. For example, we could save Executor and Taint Policy state for each trace fragment, so we do not compute the same taint propagation and instruction simulation twice. This technique has the potential to decrease execution time but at the cost of memory requirements.

Finally, further work can be done in supporting JITed languages. Support for more languages, like JavaScript (V8) or C#, needs to be provided. In addition, better control-flow extraction and more accurate memory layout mapping techniques need to be developed to more accurately analyze JIT-compiled code. Also, angr [82] provides experimental support for Java symbolic execution, which could be explored and improved upon to support the detection of Spectre gadgets in Java code.

Bibliography

[1] C. J. Anderson. One look into the future of cmos chip design. In *Proceedings of the 2009 International Symposium on Physical Design*, ISPD '09, page 1–2, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584492. doi: 10.1145/1514932.1514933. URL https://doi.org/10.1145/1514932.1514933.

[2] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8.

[3] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

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

[5] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [6].

[6] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [5].

[7] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

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

Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[9] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32648-9.

[10] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *CoRR*, abs/1807.07940, 2018. URL http://arxiv.org/abs/1807.07940.

[11] J. Horn. speculative execution, variant 4: speculative store bypas. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[12] Intel. Speculative execution side channel mitigations. https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf, 2018.

[13] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. pages 428–441, 10 2018. doi: 10.1109/MICRO.2018.00042.

[14] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[15] R. Hat. Spectre variant 1 scanning tool. https://access.redhat.com/blogs/766093/posts/3510331.

[16] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498. USENIX Association, Aug. 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko.

[17] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows, 2019.

[18] L.-A. Daniel, S. Bardin, and T. Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level, 2020.

[19] M. C. Tol, B. Gulmezoglu, K. Yurtseven, and B. Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings, 2021.

[20] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis, 2019.

[21] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus. Speculator: A tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 747–761, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359837. URL `https://doi.org/10.1145/3359789.3359837`.

[22] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz. Dynamic process isolation, 2021.

[23] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 2021.

[24] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, Nov 2019. doi: 10.1145/3319535.3363194. URL `http://dx.doi.org/10.1145/3319535.3363194`.

[25] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. Netspectre: Read arbitrary memory over network, 2018.

[26] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, chapter 4. Number 978-0-12-407726-3. MK Publications, 5th edition.

[27] C. Perleberg and A. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, 1993. doi: 10.1109/12.214687.

[28] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 284–291, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919017. doi: 10.1145/264107.264210. URL `https://doi.org/10.1145/264107.264210`.

[29] K. Kulkarni and V. Mekala. A review of branch prediction schemes and a study of branch predictors in modern microprocessors. 10 2021.

[30] G. Maisuradze and C. Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243761. URL `https://doi.org/10.1145/3243734.3243761`.

[31] A. Fog. The microarchitecture of intel, amd and via cpus. https://www.agner.org/optimize/microarchitecture.pdf, 2017.

[32] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/`

64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, .

[33] *Intel Intrinsics Guide*. Intel, https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX2, 2021.

[34] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-939133-11-3. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[35] A. Limited. *ARM Architecture Reference Manual®: ARMv7-A and ARMv7-R edition*. https://documentation-service.arm.com/static/5f8daeb7f86e16515cdb8c4e.

[36] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, Aug. 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[37] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom. Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2863–2880. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman.

[38] C. Percival. Cache missing for fun and profit. 08 2009.

[39] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack, 2016.

[40] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, Aug. 2018. USENIX Association. URL https://www.usenix.org/conference/woot18/presentation/koruyeh.

[41] P. Kocher. Spectre mitigations in microsoft's c/c++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[42] Intel. *Intel® Advanced Vector Extensions Programming Reference*. https://www.intel.com/content/dam/develop/external/us/en/documents/36945, .

[43] Intel. *Intel® Advanced Encryption Standard (AES) New Instructions Set*. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf, .

[44] J. D. McCalpin. Test results for intel's sandy bridge processor. https://www.agner.org/optimize/blog/read.php?i=378#378, 2015.

[45] A. Fog. Test results for broadwell and skylake. (2015). https://www.agner.org/optimize/blog/read.php?i=415, 2015.

[46] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. https://martinschwarzl.at/media/files/netspectre.pdf.

[47] Intel. *Intel® Software Guard Extensions (Intel® SGX)*. https://www.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-developer-guide.pdf, .

[48] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/reis.

[49] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *43rd IEEE Symposium on Security and Privacy (S&P'22)*, 2022.

[50] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408795. URL https://doi.org/10.1145/2408776.2408795.

[51] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

[52] SMT solvers in software security. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, Aug. 2012. USENIX Association. URL https://www.usenix.org/conference/woot12/workshop-program/presentation/Vanegue.

[53] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. URL https://www.fuzzingbook.org/. Retrieved 2021-03-12 11:41:11+01:00.

[54] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

[55] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. ISBN 0321486811. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321486811.

[56] F. Chow. The challenge of cross-language interoperability - the increasing significance of intermediate representations in compilers. https://queue.acm.org/detail.cfm?id=2544374, 2013.

[57] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[58] G. Project. The gnu compiler collection. https://gcc.gnu.org/, .

[59] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL https://doi.org/10.1145/857076.857077.

[60] T. Lindholm and F. Yellin. Inside the java virtual machine. 1997.

[61] L. Project. Clang: a c language family frontend for llvm. https://clang.llvm.org/, .

[62] Google. Honggfuzz. http://honggfuzz.com/, 2019.

[63] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[64] A. Davanian, Z. Qi, Y. Qu, and H. Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 31–45, Chaoyang District, Beijing, Sept. 2019. USENIX Association. ISBN 978-1-939133-07-6. URL https://www.usenix.org/conference/raid2019/presentation/davanian.

[65] F. Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, Apr. 2005. USENIX Association. URL https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[66] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[67] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014.

[68] V. Group. Fastspec. https://github.com/vernamlab/FastSpec, 2021.

[69] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[70] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new spectre era, 2020.

[71] T. Pornin. Why constant-time crypto? https://www.bearssl.org/constanttime.html.

[72] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution, 2019.

[73] C. Cadar, D. Dunbar, and D. Engler. Klee symbolic execution engine. https://klee.github.io/.

[74] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[75] R. David, S. Bardin, T. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. pages 653–656, 03 2016. doi: 10.1109/SANER.2016.43.

[76] TBD. Tbd. https://TBD, 2021.

[77] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[78] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[79] R. Team. Radare2 github repository. https://github.com/radare/radare2, 2021.

[80] TBD. Tbd. https://TBD, 2021.

[81] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[82] angr Team. https://docs.angr.io/advanced-topics/ir.

[83] S. S. Muchnick. Advanced compiler design and implementation. 1997.

[84] N. A. Quynh. Capstone. https://www.capstone-engine.org/.

[85] T. angr project. pyvex — Binary Translator. https://angr.io/api-doc/pyvex.html, 2020.

[86] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O' Reilly and Associates, Inc., 2004. ISBN 0596007124.

[87] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., USA, 1992. ISBN 0138250014.

[88] angr Team. angr/vex github repository. https://github.com/angr/vex/blob/master/pub/libvex_ir.h, 2020.

[89] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, .

[90] *Software Optimization Guide for AMD Family 15h Processors*. AMD, https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf.

[91] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/ragab.

[92] Google. Brotli. https://brotli.org, 2018.

[93] G. Wang. oo7. https://github.com/winter2020/oo7.

[94] B. A. P. (BAP). bap-toolkit. https://github.com/BinaryAnalysisPlatform/bap-toolkit.

[95] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[96] R. Team. radare2-r2pipe github repository. https://github.com/radareorg/radare2-r2pipe, 2021.

[97] R. Team. Radare2 book. https://radare.gitbooks.io/radare2book/content/, 2021.

[98] Openssl: Cryptography and ssl/tls toolkit. https://www.openssl.org/, 2021.

[99] Jsmn. https://github.com/zserge/jsmn, 2019.

[100] Http-parser. https://github.com/nodejs/http-parser/tags, 2019.

[101] libhtp. https://github.com/OISF/libhtp, 2019.

[102] Libyaml. https://pyyaml.org/wiki/LibYAML, 2020.

[103] P. Media. https://www.phoronix-test-suite.com/, 2021.

[104] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[105] Oracle. Jdk 11. https://openjdk.java.net/projects/jdk/11/.

[106] simdjson. https://github.com/simdjson/simdjson, 2021.