
CAMPUS
DE EXCELENCIA
INTERNACIONAL

**Graduado en Matemáticas e Informática**

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

# FINAL DEGREE PROJECT

# Detecting speculative information flows on large code-bases

Author: Andrés Sánchez Marín

Director: Manuel Carro Liñares

MADRID, JUNE, 2019

# Contents

# List of Figures

# Abstract

When programming, we hope that the code is executed in the order it's written, due to the optimizations on the microarchitecture such as pipelining and out of order execution, that principle doesn't hold, making the results being the same and getting a higher performance although the processor executes additional tasks.

Modern microprocessors are affected by SPECTRE v1, a vulnerability that let to exploit the hardware optimizations on the microarchitecture. It's difficult to detect the exploitable code on all its possibilities due to the so large different patterns that can exploit it.

Based on the work that SPECTECTOR does, we've leveraged the detection of the possible exploitable memory leaks by SPECTRE v1, extending the domain of analyzed programs.

New techniques for doing this work have been developed such as an analyzer for the projects internal dependencies, a more consistent parsing or the collection of statistics. We demonstrate the behavior of these methods with the *Xen hypervisor* project.

# Resumen

Cuando programamos, esperamos que el código sea ejecutado en el orden en el que está escrito, debido a las optimizaciones en la microarquitectura como el pipelining y la ejecución fuera de orden, ese principio no se sostiene, haciendo que los resultados obtenidos de la ejecución sean los mismos y obteniendo un mayor rendimiento aunque el procesador ejecute tareas adicionales.

Los microprocesadores mdoernos están afectados por SPECTRE v1, una vulnerabilidad que permite explotar las optimizaciones hardware de la microarquitectura. Es difícil detectar el código explotable en todas sus posibilidades por la gran cantidad de patrones que pueden explotarla.

Partiendo del trabajo de la herramienta SPECTECTOR, hemos conseguido aumentar la detección de los posibles flujos de memoria explotables por SPECTRE v1, extendiendo el dominio de programas analizados.

Se han desarrollado nuevas técnicas para realizar este trabajo, como un analizador de las dependencias internas de los proyectos, una traducción más consistente o la recolección de estadísticas. Se ha demostrado el funcionamento de estos métodos en un proyecto tal como *Xen hypervisor*.

# 1 Introduction

## 1.1 Background

At the beginning of 2018, security researchers discovered a security flaw in most of the modern processors since the 90's called SPECTRE [1].

Speculative execution is an optimization technique employed by modern CPUs to avoid pipeline stalls. Rather than waiting for the completion of some operation $o$, modern CPUs predicts $o$'s result and continue the execution based on this prediction. When $o$'s execution terminates, the CPU checks whether the prediction was correct. If that's the case, the CPU continues the execution. In contrast, if the prediction was wrong, the CPU reverts all changes to the CPU architectural state and restarts the execution.

There are attacks that exploit speculative execution side-effects to leak sensitive information into a CPU microarchitectural state, these are called transient execution attacks. SPECTRE is a family of transient execution attacks which affect modern processors. There are different types of SPECTRE attacks. For instance, SPECTRE v1 relies on conditional branches. It exploits the speculative execution of branch instructions (through the branch predictor) to execute arbitrary code and modify the microarchitectural state by leaving a footprint in the cache.

To reason about speculative execution attacks, researchers have proposed *speculative non-interference* (SNI), a semantic criterion of security against SPECTRE-style attacks. SPECTECTOR [2] is a tool for automatically proving whether programs satisfy speculative non-interference. The tool can be used to detect leaks of sensitive information caused by speculatively executed instructions. SPECTECTOR is implemented in the CIAO [3] logic programming system, it relies on symbolic execution, and it analyzes **x64** assembly programs.

## 1.2 Motivation

SPECTRE v1 is a hardware flaw that allows attackers to leak arbitrary memory information on any system by calling a vulnerable function with the adequate input and timing the memory access. If SPECTRE v1 mitigation is based on software, that will deteriorate the

program's performance because there will be added instructions for avoiding the leaks into the microarchitectural state. If speculation is produced in the wrong way, it will be useless because the values loaded from memory are not going to be the correct ones or if speculation is blocked, in the case it's correct. The whole process of speculation is produced by the branch predictions. So, when selecting a countermeasure we must ensure that it will stop the problem and that really is necessary to introduce it.

The ways that the code must be patched to be secure have been summarized by Miller [4]. Developers have put big efforts on mitigating the vulnerable code exploitable by SPECTRE v1, such as by isolating processes on navigators [5] or directly on OS kernels [6]. We're not sure at all that all the software that we use (and most importantly, the critical one) is patched. Some compilers have began to implement compiler level mitigations against SPECTRE on code generation as MSVC [7], Intel compiler [8] or LLVM [9]. The code exploitable by SPECTRE hasn't only an unique pattern, that's why it's so difficult to detect whether the code is vulnerable or not, as shown by Kocher with some examples of vulnerable code [10] that weren't correctly detected by MSVC.

At this point, it is necessary to have a completely reliable detection of the vulnerable code by SPECTRE v1, and which can be applied on large code-bases. Knowing that SPECTECTOR lets us to find those patterns on specific programs, so by using it we can get high efficient detection of the vulnerable code patterns, but SPECTECTOR needs to scale. By leveraging this tool to detect flaws in all kind of projects and code patterns on a short time, all code exploitable by SPECTRE v1 can be detected easily and the developers can check and fix it.

What makes this project so interesting is the fact that the software we use can be vulnerable without us knowing it, so detecting it can improve the security of the systems that we use just because the hardware that we use cannot be patched, anyway, the software can.

# 2 Background

## 2.1 Caches

Caches are small memories located between the processor and main memory [11]. Their main characteristic is that their access is faster than main memory, but also they're smaller than main memory.

Every time that a memory element is accessed, it will be stored in the cache, and in the case the element is stored already on it, its access is going to be faster. Known that the caches size is smaller than the main memory, they cannot store all the elements of main memory, but yes a subset of them.

Normally, because the caches behavior, the code executed performance is improved. The latency of retrieving elements of memory is going to be smaller in the case that the element is already on the cache. Caches cannot be accessed directly with an architecture instruction, but by measuring the time-access to one specific element, we can determine if it was previously in the cache or not.

## 2.2 Side channels

The computers we use are implementations of more high-level ideas like Turing-machines on physical systems (following physical laws). So computers are going to be affected by their implementation when doing computations, let's say, consuming electricity or emitting radiation [12]. There are other factors that involve the computations such as the latency of the operations or the latency of the memory access.

As shown, for every computation that is done, the physical system that supports it will be affected. So *side channels* are the observable elements affected by the computer's implementation that can be used to infer what computations have been done.

### 2.2.1 Side channel attacks

A system side-channel can be consulted and measured by a user not allowed to use that system. If there's the chance, by using the side channels, to obtain information about a specific information that we're not allowed to know, then we're talking about a leak.

If that information is potentially exploitable by a specific technique, we're talking about a side-channel attack. An example of side-channel attack is *flush+reload* [13], it reveals the cache contents and has been used for extracting the private encryption keys from a victim program.

## 2.3 *Speculative execution*

The *speculative execution* is a mechanism used on modern CPUs [11] that takes advantage of all their hardware. It works by predicting the outcome of the processor with microarchitectural elements (like the Branch Prediction Unit), and is used at all possible levels.

Its objective is to make the pipeline performance faster by, for example, predicting the result of a condition that hasn't been solved yet to continue filling the pipeline.

*Speculative execution* is a non-controllable feature on the processors, its behavior changes constantly with the system operations. When executing instructions on the pipeline before they must be really executed, it maximizes the usage of the processor functional units.

When doing speculative execution and there's a branch condition, the processor may decide what of the possible branches is going to be executed (task performed by the Branch Prediction Unit). Speculative execution will continue on the predicted branch by modifying the microarchitectural state. When the condition of the branch is finally solved, it's going to be checked if the prediction was right or wrong.

In the case prediction was right, all the computations performed speculatively are going to be committed (set up at architectural state). In the case it was wrong, all the information related with the computations done speculatively will be dropped, but microarchitectural elements (i.e. the caches) that have been affected by that computations, will preserve their state.

For an observer, there won't be any differentiation because the final result doesn't change, but the performance is much greater when using the maximum possible of the processor.

To stop speculative execution, Intel and AMD recommend the use of the LFENCE instruction. It serializes the memory operations to be performed, this way, the speculation is stopped in the case there's going to be a value loaded from memory, so caches won't be altered during

speculation.

## 2.4 SPECTRE attacks

SPECTRE attacks involve inducing a victim to speculatively perform operations that wouldn't occur during the correct program execution and which leak the victim's confidential information via a side channel to the adversary [1]. This is done by performing instructions speculatively that may lead to side effects on the microarchitectural state, i.e. loads from memory.

SPECTRE variant 1 attack exploits the conditional branches by mistraining the branch predictor, causing the CPU to violate the program semantics by executing code that wouldn't be executed otherwise. If the executed code produces side-effects that can be observed, then, there is an information leak. The attack consists on 4 steps:

- Train the branch predictor
- Fill the cache with prepared data
- Exploit a vulnerable function (by calling it with the adequate input) to leak the desired element
- Infer which element has been accessed by timing the access to the cache lines with the same data that we prepared

The previously defined steps can be customized by several ways (depending on how the same vulnerability can be exploited). An example of vulnerable code is on Figure 2.1, that code gets compiled to the one on Figure 2.2.

```
1   if (y < size)
2       temp &= B[A[y] * 512];
```

Figure 2.1: SPECTRE variant 1 — C code

On line 4 of Figure 2.2, the branch predictor is able to perform a misprediction and make the program continue to line 5. Then, based on the element located on `A(%rbx)`, there will be another memory access that leaves a footprint on the cache. With the second access, an attacker can infer what's located on `A(%rbx)` by measuring the access time to all the elements of B. For this attack, the attacker needs to fill the cache with arbitrary data, so when measuring

```
1       mov  size, %rax
2       mov  y, %rbx
3       cmp  %rbx, %rax
4       jbe  END
5       mov  A(%rbx), %rax
6       shl  $9, %rax
7       mov  B(%rax), %rax
8       and  %rax, temp
```

Figure 2.2: SPECTRE variant 1 — Assembly code

the access time to the elements of the cache after calling the vulnerable code, the longest time will correspond to the data that's on the memory position specified.

When SPECTRE v1 is exploited, it doesn't matter if the first memory access is been performed, it won't give enough information about what has been loaded. That's why we need to rely on an second action with the loaded value, like a second memory access.

This is not the only way that the exploit can be done, there are so many patterns as the ones suggested by Kocher [10] or the different variants that have appeared [14] [15] [16], also there are approaches that don't rely on the memory hierarchy [17]. To mitigate the vulnerability on the code there have been proposed lot of strategies like creating dependencies between the branch condition and the values used to access memory [9], manually isolate the executable code [5] or directly stop the speculation by architecture instructions [18]

### 2.4.1 Mitigating SPECTRE

When a countermeasure is placed on a program, its performance will be deteriorated. It's much more important to know if a countermeasure must be really inserted rather than use it in all the possible cases. Therefore can be cases where a countermeasure that is not necessary is used or where a countermeasure should be exist the compiler don't insert it.

The compilers need to analyze the program to be compiled to see where it's necessary to insert the countermeasures. The methods used to detect it may be efficient, so it's important to detect if the countermeasure is mandatory or not.

### 2.4.2 Methods for avoiding SPECTRE variant 1

#### Stop the speculation

As explained in 2.3, the LFENCE instruction stops speculation, so by injecting it, SPECTRE v1 can be avoided. According to the Intel manual, the behavior of the LFENCE is the next: *"Performs a serializing operation on all load-from-memory instructions that were issued prior the* LFENCE *instruction. Specifically,* LFENCE *does not execute until all prior instructions have completed locally, and no later instruction begins execution until* LFENCE *completes"*. So this instruction must be inserted between the comparison instruction and the one that leaves the footprint.

#### Speculative load hardening

The *speculative load hardening* (SLH) is a method proposed on the LLVM mainline by it's developers [9], and naturally can be used by CLANG. It consists on hardening all registers that contain memory information, whenever they have been loaded from memory and are going to be used to perform another action afterwards.

During the hardening, there will be a predicate state that represents the current state of speculation (if the prediction done is wrong or not) by its value. It's going to be on a register and it will be all ones in the case the speculation is wrong, all zeroes otherwise. The hardening is done by a bitwise `or` operation with the current predicate state, which determines whenever there's a wrong speculation. On Figure 2.3 there's an example of the code of Figure 2.2 hardened.

The predicate state value will change with the `cmov` operation, in the case a speculation is being performed, the cmov will set the mask register to all ones, so the next bitwise `or` operations with that mask will set the destiny operand at all ones.

In the case the whole exploit is performed on a different function than the one where the branch predictor has been used, the predicate state will be lost between function calls. For avoiding it, the predicate state is preserved in the stack pointer, so there are another 2 additional code patterns (represented in Figure 2.4) to retrieve the predicate state and to store it.

#### GCC method

On the GCC mail list, a method for mitigating SPECTRE v1 was proposed [19] with the respective changes that GCC source code may have to implement it, although it hasn't been

```
1     mov  size, %rdx
2     mov  y, %rdi
3     cmp  %rdi, %rdx
4     jbe  END
5     cmovbe  %rcx, %rax
6     lea  A, %rcx
7     movz (%rdi,%rcx), %ecx
8     shl  $9, %rcx
9     orq  %rax, %rcx
10    lea  B, %rdx
11    mov  (%rcx,%rdx), %cl
12    or   %al, %cl
13    and  %cl, temp
```

Figure 2.3: Speculative Load Hardening on SPECTRE variant 1 — Assembly code
On line 5, set the predicate state according to the outcome of the prediction
On line 9 the loaded value is hardened

```
1     # To retrieve the predicate state
2     mov  %rsp, %rax
3     sar  $63, %rax
4
5     # To set the predicate state
6     shl  $47, %rax
7     orq  %rax, %rsp
```

Figure 2.4: Predicate state tracing — Assembly code

implemented yet on the mainline distribution. Like in SLH, it consists on using a mask register, that will be set up by a subtraction with borrow (`sbb`) after the comparison used for the branch operation. Based on the comparison result, the mask register will be set at all ones or all zeroes, and with that value we will perform both the jump operation and the mask. The mask register will harden the index of the first memory load by a bitwise `and` operation after the branch instruction is performed to always load the same element.

```
1    mov   array1_size(%rip), %eax
2    cmp   %rax, %rdi
3    sbb   %rax, %rax
4    test  %rax, %rax
5    je END
6    and   %rax, %rdi
7    movzb array1(%rdi), %eax
8    sal   $9, %eax
9    clt
10   movzb array2(%rax), %eax
11   and   %al, temp
```

Figure 2.5: GCC working mitigation on SPECTRE variant 1 — Assembly code

### How the proposal can be exploited

In the case of 3 nested memory accesses, this mitigation won't be totally effective because the first value obtained from memory is the only one going to be hardened (in the case of bad speculation). Due to that, the value obtained in the second memory access can be different between 2 executions, that produce a memory leak on that value by doing the $3^{rd}$ memory access.

```
1   if (y < size)
2       temp &= C[B[A[y] * 512] * 512];
```

Figure 2.6: SPECTRE vulnerable pattern against GCC mitigation — C code

## Vulnerable programs against compiler mitigations

Later, in 5.2, we will present vulnerable programs against this mitigations that the compilers implement, making them not able to detect the exploitable code, nor to mitigate it

correctly.

## 2.5 SPECTECTOR

Presented on 1.1, SPECTECTOR [2] analyzes **x64** assembly programs to check if there are information leaks introduced by speculatively executed instructions or if the program is leak-free.

SPECTECTOR symbolically executes the program under analysis with respect to a semantics that accounts for the speculative execution's effects. During the symbolic execution, SPECTECTOR derives SMT formulae characterizing leaks caused by speculatively executed instructions. It then relies on the Z3 SMT solver to determine the presence of possible leaks.

SPECTECTOR helps the developer to detect if it's necessary to inject a mitigation or not because it uses formal methods for detecting automatically the vulnerable code by specifying the vulnerable gadgets. It can be also extended to work with any language always by having translation from it to the internal representation ($\mu$ASM). An example of this is the compilers internal representation, like LLVM IR, this way, the SNI checking can be done at compile time.

# 3 Algorithms

## 3.1 Analysis algorithms

For verifying that a program holds a certain property (in this case the *speculative non-interference*), we must explore all its possible executions to verify it. This task is accomplished by SPECTECTOR by using concolic execution [20].

Concolic execution works by executing the program symbolically and concretizing the values when obtaining a full path. Any operation that may change a value is going to be taken as a restriction, which will be imposed over the symbolic values. When reached the program conditional branches, the restriction is going to be the one which asserts the branch condition (or negates it). On Figure 3.1 a diagram about the concolic execution is represented and how the branch conditions affect it.

When analyzing the program, it's assumed it's going to be well-constructed, let's say, any other leak no produced by SPECTRE variant 1 won't be detected.

### 3.1.1 Checking speculative non-interference

For verifying SNI, is required that the program doesn't leak more information executing speculatively than non-speculatively. Formally, whenever two indistinguishable configurations produce the same non-speculative observations, they must also produce the same speculative observations.

For checking SNI, we're going to distinguish between 2 different types of possible leaks, depending on what is done with a speculative loaded value: if it's used to do a memory access (checked by MEMLEAK process) or if it affects the program control flow (checked by CTRLLEAK process). The checking procedures and all the formal definitions, can be found defined on the previous work over SPECTECTOR [2].

For checking the SNI, the formulas are encoded on SMT language, SPECTECTOR interacts with an external SMT solver for solving them. Each time a complete path of the program is obtained, the process of checking the SMT formulae is done.

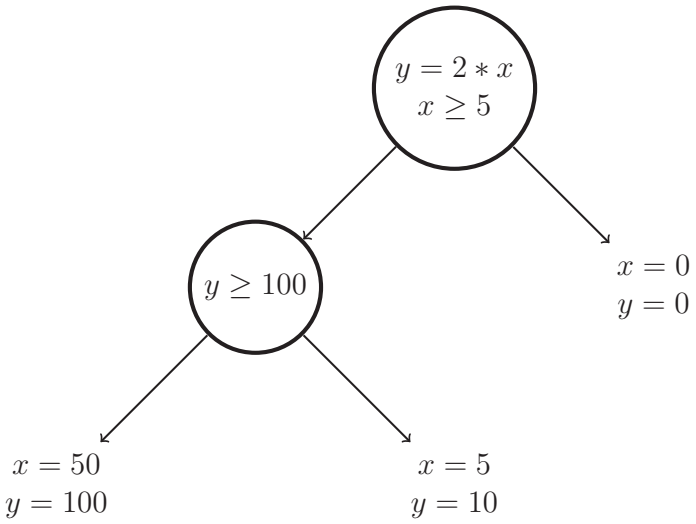



Figure 3.1: Concolic execution diagram
Whenever we get to a condition, the concolic module will determine the 2 possible paths that are part of it (if possible), they're only concretized when the execution is done. For each branch condition the assertion is going to be the left path and the negation the right one.

### 3.1.2 Always-mispredict semantics

To reason about how the speculation is done on the microarchitecture, SPECTECTOR gives an abstract model of it. The speculation is interpreted as a whole structure of nested states, a new level on the structure is created each time that a speculation begins. After the resolution of the branch condition, it can be either rollbacked or committed. In the case of rollback, the information obtained during the speculation will be dropped and the state will be the one before beginning the speculation. In the case of committing, all the information obtained during the speculation will be set up on the parent level of the one that has been modified in the data structure.

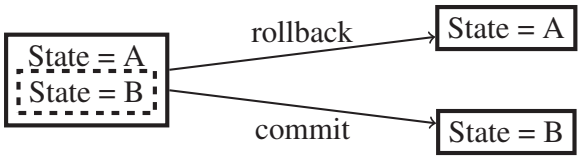



Figure 3.2: Rollback & commit diagram
The nested state (dashed box) will be dropped by a rollback or set up by a commit, while the upper state won't be altered

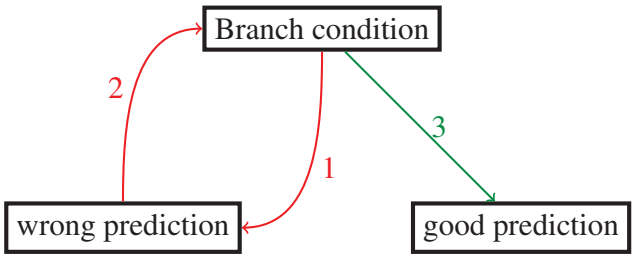



Figure 3.3: Misprediction diagram
Each arrow is labeled with it's execution order, after the misprediction branch finish, we return to the branch condition and continue with the correct prediction

When there's a branch instruction, the processor uses the branch predictor [11] to determine what's going to be the outcome. In the semantics of SPECTECTOR, we've determined that this mechanism will always produce the worst possible case, this way, we're able to explore all the possible paths.

If a state is nested, its speculation window will be initialized with the value of the one that state its allocated on. When decrementing the speculative window we're only going to decrement the correspondent value of the most nested state. As mentioned, a state can nest another state, which at the same time can nest another state and so on...

## Speculative window and backtracking

There's a mechanism called *speculative window* that represents the number of instructions the program can perform speculatively.

Whenever a condition is chosen, the speculative execution will commence and the *speculative window* counter will be decremented at each step until the end of the program is reached or the *speculative window* cannot be decremented more. When a complete path is obtained, SNI is checked with the produced trace and in the case that is verified it explores the next path by rollbacking to the last bifurcation point.

The rollback is done because the selected condition was wrong, so it recovers the state of the last branch condition and chooses the correct path (the Figure 3.3 represents this behavior). In the case that a branch condition is found again, it performs the same (nesting the speculations), always taking care of the restrictions. In the case the prediction is good, instead of doing a rollback, it will commit, setting all the information obtained from the speculation into an upper level of execution.

## 3.2 SPECTECTOR architecture

The Figure 3.2.2 represents the design chosen of the whole process architecture. It consists on different modules with a data flow between them, starting with the program to analyze, that is served as an input to the parser module. Then, the resulting $\mu$ASM program (and some additional specifications) is analyzed to the analysis module which follows the concolic execution method for finding speculative leaks.

### 3.2.1 Translation module

To work with real assembly programs, it's needed a frontend to translate all the instructions from the assembly specification they come from to the one defined on SPECTECTOR. The current supported assembly **x64** syntax are the GNU Assembly Syntax (GAS) and the Intel syntax customized by Microsoft.

The translation is done on polynomial time relative with program's size because it's a mapping $1 : n$ instructions at the same time that it fills some other data structures with directives contents, something we're going to explore later on 4.1. For each one of the supported instructions there is defined a translation to the $\mu$ASM specification. This module may consider all the specifications of the processors, so when translating the program, the $\mu$ASM resulting program may perform all the not explicit operations (i.e. side channels).

### 3.2.2 Analysis module

When analyzing the $\mu$ASM program, as described on 3.1, the concolic execution is needed.

To determine how the $\mu$ASM behaves (and how the program should continue), there's a specification of the $\mu$ASM semantics. This module evaluates the $\mu$ASM instructions by following their specific syntax. Some instructions of the model are: `beqz`, `spbarr` and `load`. In the process, the registers, values and expressions (arithmetic, bitwise...) are evaluated.

The program will be conformed by its instructions but also by the memory and registers which are going to be modified during the semantics evaluation.

At the same time than the program is evaluated, the SNI is checked for each path reached. The analysis process begins by asking for a trace and then running concolically the program (following the specified semantics). If the selected path is detected as insecure, the process ends showing the counterexample. Otherwise, if all the paths are detected as safe then so will the whole program.

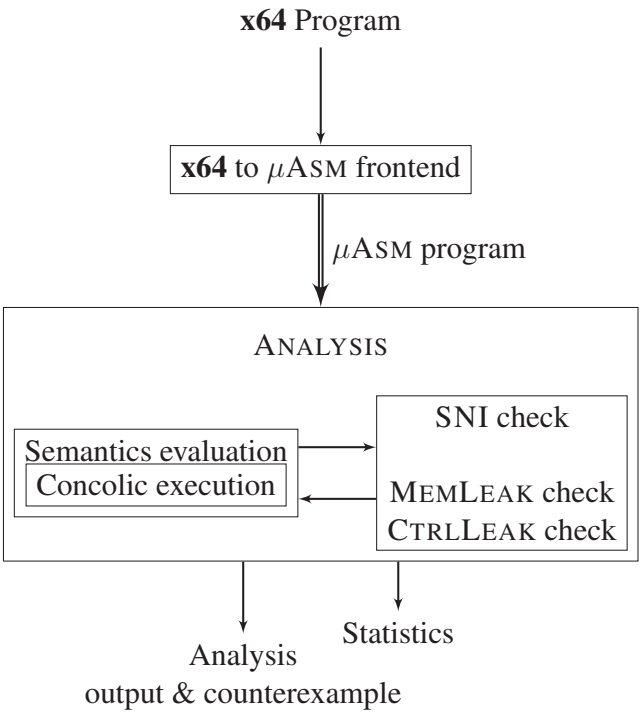



Figure 3.4: Diagram of SPECTECTOR architecture

The input program can be either a x86_64 assembly or a $\mu$ASM one, it's passed to the $\mu$ASM translator module. The program translation and the specific semantics from the lenguage are going to be passed to the analysis module. The analysis module is going to produce the traces for the program based on the speculative semantics, and for each trace it's going to theck that verifies the SNI (by the 2 processes MEMLEAK and CTRLLEAK). To produce new traces, the analysis module relies on an external concolic module, which generates the traces by using concolic execution of the program. The whole analysis will end in the case a leak is found or all the traces satisfy SNI. Together with the result of the analysis, are going to be the statisitcs produced by the process.

## 3.3 Model specifications

When designing a tool for performing tasks, it should follow some specifications. SPECTECTOR has been adapted to follow the current specifications according to memory, registers contents or functions calling conventions. Also, there are too complex or unknown specifications that may be obscure, like branch predictor outcome. These unknown behaviors will only modify the procedure followed to obtain paths, the checking procedures will remain the same. If a different specification (i.e. prediction oracle) is needed, then the program should be executed under the new specification.

### 3.3.1 Calling convention

Due to that what we're analyzing are functions, it may be known automatically which inputs can be controlled by the user and which ones are defined. It's also necessary to make a distinction between the data that won't be changed during all the execution and the one can be changed. Later we will explore this cases on 3.3.2.

Each compiler is going to manage its own calling convention (i.e. Microsoft [21] or LLVM [22]). This may change on system optimizations, i.e. the values can be given through stack relative memory addresses.

In the case the resulting program is well-written (the compiler doesn't mess up on some call parameters), SPECTECTOR will execute as the normal program only with the difference that the values are going to be interpreted symbolically in the case they don't have a concrete value. So when dealing with calling conventions, the semantics are going to follow program ones during the execution, anyway the values are initialized or not.

### 3.3.2 Memory policies

There are two cases: the data located in memory is static (it cannot change during the whole execution) or dynamic (its contents can be modified during the whole execution)

```
1    mov  y, %rbx
2    mov  A(%rbx), %rax
```

Figure 3.5: Variable data structures on SPECTRE variant 1 — Assembly code

On Figure 3.5, we assume `y` to have a inbounds value (on `A`). So if `A` is filled with static

data (all the data on `A` won't change across the execution), the possible values that are going to remain on `%rax` are going to be set during all the execution. Otherwise, if `A` can be modified, when indexing it, `%rax` is going to be either symbolic or on a set, depending on how the modification of `A` values come from.

To determine if the data is going to be static or dynamic, that information must be provided on execution time to keep some memory locations uninitialized (with symbolic data). In the case that the input is associated with some memory location, we may also know the policy for its contents and the size that the location has.

### SPECTECTOR way of managing the policies

We need to go further when defining policies on SPECTECTOR, we explore speculatively different traces and find a difference between two traces. So we need to provide the information about which data can vary between traces and which not.

SPECTECTOR is designed with two possible policies for the memory locations, HIGH & LOW.

- HIGH policy specifies that the contents of the memory address is symbolic and can change between 2 different assignations meanwhile.
- LOW policy represents that the contents of a memory address may not be changed between 2 different assignations during the execution.

The non symbolic values are going to be always LOW (logically), and the symbolic ones are going to be HIGH. The registers are going to be always LOW because we consider that during the execution they can be observed.

The user can specify the contents of the memory and registers by giving them an initial value (setting them automatically to LOW) or specifying that are going to be LOW (only in the case of memory directions).

## 3.3.3 Sizes on the operations

The instructions on the **x64** model are designed to work with different sizes. There are registers mnemonics which reference to different subsets of the same register (such as %eax and `%al`, which are subsets of the register `%rax`). The operations mnemonics also implement this feature to work with different objective sizes.

SPECTECTOR model can be extended to get support for managing the sizes both in the frontend and the analysis module. This work goes further from this thesis one.

## 3.4 Large code bases analysis algorithms

SPECTECTOR is designed for analyzing functions, in the case the one is going to be analyzed involves huge traces with complex path exploitation, then, the analysis is going to be much harder. The checking of SNI needs from the speculative execution, so in the absence of branch conditions, the program is going to be always secure.

A classical argument against precise static analysis (such as the one done by SPECTECTOR) is the scalability, more precisely the issue comes out from huge structures that may be interpreted at runtime like loops. Anyway, this can be solved by detecting different patterns the code may have and abstracting over them. This method can be performed either in the tool side by implementing detection models, or specified by the user on the input parameters.

To help the user to detect this patterns easily, we've developed a module that generates statistics of the execution at runtime which we're going to talk about later on 4.3. This way, the code that is executed several times for a specific concretization of the configuration will be detected easily, letting to analyze it separately.

# 4 Implementation

## 4.1 Translation of assembly programs

As explained in 3.2.1, SPECTECTOR has a translation module for creating the $\mu$ASM programs.

This module has been adapted to implement all the new features supported by SPECTECTOR such as the processing of the memory elements defined in the assembly program and their storage on the heap.

### Instructions mapping

SPECTECTOR has a defined model of instructions to work with, similar to the microarchitecture actions performed by a normal processor. All this information is defined on the semantics of SPECTECTOR, so every **x64** program translated to $\mu$ASM must have defined this instructions.

For this task, the instructions on the **x64** specifications have been translated to a set of $\mu$ASM instructions which will behave the same than in **x64**. An example of this are the **x64** instructions `push` & `pop`, which are going to be translated to a set of $\mu$ASM instructions. Both instructions receive one argument which is going to be in the case of `push` the value to store on the stack and on the case of `pop` where's going to be stored the retrieved value from stack.

We need so to specify the necessary standards from the **x64** model for doing these instructions work on the $\mu$ASM semantics. The specifications are the following:

- A specific register for the stack pointer which contains the value of the last stored value on the stack
- A masking operation over the stack pointer register because in the **x64** model there are only used the 48 lowest bits of the stack pointer

So the resulting transformations are the following (considering the translation of `%rax` to be `ax`):

$$\texttt{push \%rax} \Rightarrow \begin{cases} \text{sp} \leftarrow sp - 8 \\ \text{store(ax, sp} \wedge 0x0000f \ldots f) \end{cases}$$

$$\texttt{pop \%rax} \Rightarrow \begin{cases} \text{load(ax, sp} \wedge 0x0000f \ldots f) \\ \text{sp} \leftarrow sp + 8 \end{cases}$$

A similar work has been done with all the currently supported **x64** instructions, by looking on they behavior at low level and introduce their translation to $\mu$ASM instructions.

## Memory elements

The assembly programs have defined the instructions and the directives, the directives give additional information about the program no related with the execution of the instructions. The memory contents are given by the directives: the name assigned to the memory sections, the size they take and their initial values. This data is processed by the assembler program, setting up the memory correctly with the instructions. As an assembler program, we do the same with SPECTECTOR's frontend, using the directives for setting up the $\mu$ASM program's memory.

SPECTECTOR's frontend can ignore specified memory contents (only by not tracking the specified ones), receiving the no initialized memory objects an an input with the program. On the analysis side are going to be received the translated program and the memory contents initialized. These memory contents will also contain the symbolic direction for each one of the variable names specified on the original assembly program.

The stack is used to carry information across functions, it stores values that can be retrieved. In our model, the stack is used for this committee, using the stack pointer and base pointer registers, that are modified with stack operations (such as `push` & `pop`).

## Operands

A same instruction with a same mnemonic can have different operation codes because its operand sizes. **x64** programs implement different mnemonics for the operands and for the operations, depending on the sizes the operands must have.

We process that data on the translation module of SPECTECTOR, this way, we preserve the operand sizes.

Following the specification, operands also can be static values such as numbers or registers used for indirect jumps (which are prefixed by $*$).

## 4.2 Project structure

SPECTECTOR is designed for analyzing assembly files. The projects are composed by several files and libraries that have their own dependencies, each one of those dependencies can have at the same time more dependencies and so on. After all the compiling process, we obtain the desired files (i.e. binaries) of the project. So it's necessary a method that analyzes the whole project structure and generates its dependency tree.

### 4.2.1 Projects linkage

The generation of assembly files of a project may have some issues because most of the references are missed (i.e. calls to functions of another files). This problem is solved at build time by the linker tool that the OS provides.

We've developed a method for solving all the inner (between the project files) dependencies on projects. For achieving this aim, we rely on external tools of the LLVM project [23], so every part of the code was compiled to LLVM bytecode for this purpose.

The process followed for this purpose is the next:

- Compile all the project to LLVM bytecode and collect it by using CLANG (or any other compiler dependent on LLVM that can generate the LLVM bytecode)
- Collect the correspondence between functions and files on a data structure (using `llvm-nm`)
- Identify the missing labels of each file and detect the files they correspond to
- Recursively solve the dependencies by linking the LLVM bytecode files (using `llvm-link`)
- Generate the assembly files from the LLVM bytecode ones (using `llc`)

A file cannot be linked with a file that contains at least one of its labels, otherwise, there would be a conflict when accessing to that label.

Also, all the linkage can be done on a single file, but generating a file that contains all the project code, which obviously is going to take more space than the small programs. We're going to explore this case on 4.2.3.

We can think on the linkage operation as a function that receives 2 inputs and gives as output the resulting program with the 2 inputs linked, it holds the commutativity (2 programs

are going to produce always the same output independently of their entry order) and associativity (to link `A` with `B` and then with `C` is the same than linking `A` with the output of the linkage of `B` and `C`).

With the operation of linking, the programs are isomorphic to Map structures with a key `label` and a value `content`, where `label` is the label in the code which can be referred, and `content` is the code indexed by that label. When linking, it must hold that the intersection of the labels between the two programs must be empty, if so, the 2 maps appended are the result of applying the linkage function.

All the operation of solving dependencies is designed for solving the internal dependencies on a project. In the case there are also external dependencies, the files that contain them can be added to the folder where we perform the whole process.

### Structures for the linking

During the linkage process, we use 2 data structures, one with the correspondence between the labels and the file they belong to, and the other one with the files that a specific file depends on. The second data structure is filled by looking the label dependencies for each file and consulting on the firs data structure which file corresponds on.

These data structures are filled during all the process of linkage, the diagram of their representation can be found on Figure 4.1

| | |
|---|---|
| Label A | File 1 |
| Label B | File 2 |
| ... | ... |
| Label X | File 2 |

| | |
|---|---|
| File 1 | File 2, File 4 |
| File 2 | File 4 |
| ... | ... |
| File N | File 2, File 3 |

Figure 4.1: Linkage data structures used

## 4.2.2 Tree of dependencies

There can be cases where a specific file depends on several files and at least 2 of these files depend on the same file. When that occurs, we cannot link twice the same file, so there would be a conflict.

Formally, if a program `A` depends on programs `B` and `C` and, at the same time, program

B depends on program C, the program A needs to be linked with original programs B and program C, otherwise, by linking A with a linked version of B and C, it will generate a conflict because the definitions by C are already present on the linked version of C.

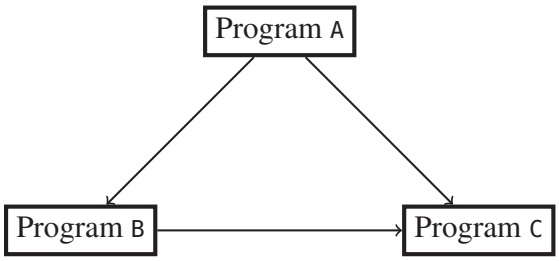



Figure 4.2: Dependency conflict diagram, the arrow symbols dependency

For solving this problem, we don't rely on linked programs for doing the final linkage, the linkage is done with the original files, this way, we avoid the problem of duplicate labels. Another possibility is to link the desired file with files already linked but keeping track of what files they link to. In this case, the linkage process would need to determine which files are the missing ones for the final result and always preserving that there will be no overlap between existing functions, so this solution introduces unnecessary over-costs.

### 4.2.3 Link all files on a single one

The method where we solve all the dependencies for each file applying that process for every file is useful. In the case that a whole project is going to be analyzed, getting a single file for the whole project is going to be better because it will help to avoid all the redundancy produced by the linkage of the same file for several times on different produced files.

We apply this method with the *Xen hypervisor* project (which we're going to talk about in 5.3), in which we're analyzing the whole project at once (instead of a specific function). If we apply the method that produces an assembly file for each original file, the whole set of files is going to be of 1.2G, and when we produce a single file it's only 15M.

## 4.3 Collection of statistics

We've added to the code a instrumentation for getting fine-grained statistics of the analysis process, we consider them necessary just to detect what are the cases where SPECTECTOR takes more time working on and by that way, remove the most bottlenecks as possible.

## 4.3 Collection of statistics

The code has been instrumented adequately for collecting the statistics of the whole execution, such as the number of instructions passed through, the length of the traces, the time taken to check the SNI or to generate a trace.

All the statistics are collected and whenever the process terminates, writes a `json` file with that statistics. The `json` notation has been selected because its capability of allocate `json`s as attributes, also because an existing library of CIAO that encodes and decodes `json` files. The statistics can be classified on 3 different hierarchies: path dependent, analysis dependent, execution dependent. This classification is deeply related with the way that SPECTECTOR executes, so in the whole execution, the different statistics will be considered into one hierarchy or another depending on the state of the execution.

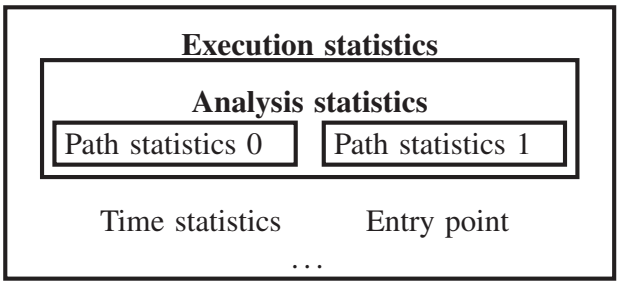



Figure 4.3: Diagram of stats structure

The statistics include the next information:

- Path statistics
  - Length of the trace executed
  - Statistics retrieved from the concolic module
  - Time taken to generate the trace
  - Time taken to check the SNI
  - Status of the path
  - Number unsupported instructions found
  - Undefined labels found
  - Steps done over the trace
  - pc's accessed and number of times
  - pc's of indirect jumps found

- Analysis statistics
  - Number of paths explored
  - Statistics of all the paths analyzed
- Execution statistics
  - Function analyzed (name)
  - Total time of the execution
  - Time taken to translate the input
  - Status of the whole execution

The resulting statistics are going to be written on a `.json` file. If the process analyzes the program with more than one entry points, the statistics are going to be written on different `json` objects.

The Figure 4.4 is the resulting `json` after the analysis of the $8^{th}$ Paul Kocher example compiled with CLANG without applying any mitigation (UNP) and with the optimization flag `-O0`.

To generate and keep track of the statistics we've used the `datafacts` module provided in the CIAO library. It gives us the possibility of manage the data in execution time without the necessity of tracking it across all the execution, so when necessary, we modify or retrieve this data.

The interface designed to assert new statistics is the following: one predicate fir dropping the information about an arbitrary fact and another predicate for asserting information on an arbitrary fact. To manage all the statistics it has been used an unique data fact denoted as `list_stats/2`, which has as the first argument the type of statistic to trace and as the second the statistic that we want to consult, assert or retract.

This model is easily extensible, every new type of statistic can be asserted by adding a predicate for initializing it and another for asserting new statistics. A example of it is the next one, showing the unique 2 necessary predicates for adding path dependent statistics.

```
1 % For restarting path stats
2 init_path_stats :- retractall_fact(list_stats(path, _)).
3 % For adding a new path stat
4 add_path_stat(Stat) :- assertz_fact(list_stats(path, Stat)).
```

Figure 4.5: Statistics interface — Prolog code

```json
{
  "file": "results/out/clang.08.lfence.o2.s.json",
  "paths": {
    "length": 1,
    "0": {
      "pc": {
        "18446744073709551615": 1
      },
      "unsupported_ins": 0,
      "unknown_labels": [],
      "indirect_jumps": [],
      "steps": 21,
      "formulas_length": [],
      "status": "safe",
      "time_trace": 0.4349999999999952,
      "time_solve": 0.2409999999999997,
      "trace_length": 6,
      "concolic_stats": []
    }
  },
  "entry": "victim_function_v08",
  "status": "safe",
  "total_time": 2.063000000000002,
  "time_parse": 0.6589999999999989,
  "name": "target/clang/08/lfence.o2.s"
}
```


Figure 4.4: Statistics of CLANG `-O2` for Paul Kocher example 08 — `json`

Also, this functionality can be extended to produce statistics not only in `json` format, only by changing the way that the final statistics are produced (because the part where the `json` file is produced is at the end of the process), and also it can produce pure logic programming predicates that represents the statistics.

### 4.3.1 Visualization of the results

After running SPECTECTOR over a whole database, the statistics produced can be used to determine where does the tool needs to improve its performance. When determining which SPECTECTOR features must be improved, we don't distinguish between the project or set of files that we're analyzing because it won't contribute in a significant way to the overall result. The whole execution will help to determine where does the tool to improve its performance.

For that committee, the statistics produced on the different executions can be easily visualized by graphs or tables. Both models work by loading the necessary files for showing the statistics.

#### Tables visualization

A tool for visualizing the tables with the results on the different timeouts has been developed, it only relies on a general file which main contain all the `json` produced after the end of the whole analysis. On Appendix B there is an example of the table visualization and some other features this visualization provides.

It provides a whole overview of the status result of the projects and the individual stats for a project. This visualization is more focused on the project analysis rather than on the possible improvements over SPECTECTOR. This visualization introduces mechanisms for toggling the visualization of custom elements.

This tool has been developed using web technologies, so a HTML skeleton is filled with the data of a specific `json` file that follows the specification and processed by a JavaScript program using the DOM. Thanks to this technology, the visualization can be fully interactive.

Due to the huge data that the statistics provides use, the visualization can be done in several ways, the table visualization implements very simple ones.

#### Graphs visualization

This visualization is focused on showing the scalability that the tool offers based on the results obtained.

It offers information about specific statistics and their correlations by plotting graphs (produced using matplotlib [24]). To create the graphs it's necessary to use the `analysis.py` script with the adequate parameters. These parameters should indicate the outputs of the analysis and some additional information such as the lines of code.

The plots give a whole overview of the project analysis with SPECTECTOR. For example, by showing the time they the solver takes with any other stat we can detect easily which stat is correlated with slowdowns.

## 4.4 Internal timeouts

To deal with possible non terminating executions (or very long ones), a mechanism of timeouts has been implemented. It consists on different types of timeouts that the user can specify to deal with the type of programs described before. The timeouts specified are going to be dependent on the different status of the execution.

To terminate the whole process in a specific time, there are 2 different ways. A "hard" timeout can be set externally to SPECTECTOR process, it will kill it in the case that it doesn't terminate in the given time. This "hard" timeout won't generate the statistics information. This can be done with tools like the UNIX utility `timeout`. A "soft" timeout is set on the process, that in the case that SPECTECTOR continues working when the timeout jumps, it writes before the statistics before dying.

Another 2 different timeouts have been implemented. One is the SNI check timeout, that can be set using the SPECTECTOR command line flag `-noninter-timeout` for avoiding the process to get stuck during the execution because a hard formula. Those timeouts can be passed to **Z3**, so whenever they're reached, we're going to receive a different state that denotes it.

Whenever a timeout is reached, the statistics of the process will show that information. For example, if the SMT checking timeout is reached during the evaluation of speculative non-interference, the status of that check will be set as unknown and the program will continue as if it's safe, the data will be shown on the stats and later can be post-processed.

## 4.5 Leaks checking

As explained on 3.1.1, for detecting speculative leaks we rely on SNI, which is checked under 2 different process: MEMLEAK and CTRLLEAK.

Knowing that the leaks based on memory are generally easier to exploit because they rely

on the cache behavior to store the desired data. On control leaks there are more complex structures to exploit, but the also may produce difficult structures to exploit (it all depends on the code executed by the different condition). As explained on [2], Paul Kocher example number 10 represents a control leak. On this example, the second load is not symbolic (it's going to be always the same), but it's performed under a second condition, so the code executed in this second condition is the load, which will help us to infer what was the first value.

All it depends on what we do with the first symbolic loaded value, another memory access or a control action that can trigger more actions.

We give the opportunity of checking independently either the data or the control leaks y passing the flags `-only-data` and `-only-control` respectively to SPECTECTOR.

### 4.5.1 Configuration files

SPECTECTOR receives input parameters (entry point, starting heap direction, LOW memory directions...). There are default values set for that parameters int he case they aren't specified. For not giving the parameters in the call, there can be given by configuration files, that for consecutive calls that need the same parameters, reduces the call complexity. By default, the configuration file taken will be the one on the same folder that the file to analyze and named as `config`.

For the examples where we need to specify this information, the method of specifying the input parameters at once (and setting them for later experiments) is going to benefit us. The configuration files are files composed by `prolog` assertions, this way, is going to be much easier to get an integration with the main process which is also written in `prolog`.

# 5 Case studies

After implementing all the changes described in previous chapters, we've developed a system to perform the benchmarks and to

To test the tool performance, we've run it over the next test suites:

- Paul Kocher examples [10]
- Some custom examples, based on the functionality of the Paul Kocher's ones
- XEN *hypervisor project* [25]

These tests have been performed to show the improvements on the tool. The Paul Kocher examples have been performed to see the difference in the runtime between versions, the custom examples to show how the new mitigations can be analyzed, the XEN *hypervisor project* to show how able we are to analyze large projects.

## Benchmarks system

2 systems have been enabled for running the experiments:

- A virtualized environment under QEMU 2.5+, OS Debian/GNU Linux 9.0 64 bit, 64GB DDR4 RAM (2666 MHz), Xeon Gold 6154 @ 3GHz w/ 72 virtual cores
- A device running the Arch Linux 5.0.0 Linux kernel, CPU: Intel(R) Core(TM) i7-7560U CPU @ 2.40GHz 4 cores, 16GB LPDDR3 RAM (2,133 MHz)

The compilers used for creating the assembly files from the source code are: CLANG v8.0.0, ICC v19.0.0.117, GCC v9.0.0, VISUAL C++ v19.15.26732.1 & v19.20.27317.96.

For running the experiments, a SPECTECTOR process is launched for each one of them. In the case of a large project (such as XEN *hypervisor project*), we only use an only compiler (as described in 4.2). Otherwise, we previously compile the source files of the small examples with all the compilers we're going to analyze.

For running the experiments, we've used the `GNU parallel` tool, which let us several experiments in parallel. Its use lets us to take advantage of the systems with several CPU cores

that otherwise cannot be used. The way that the parallelization is done with this tool is only by launching several process without no relation between them, is a mechanism only to speedup the analysis when dealing with several experiments.

Also, SPECTECTOR hasn't yet the potential of analyzing all the programs. There are some instructions that aren't implemented yet on the parsing side nor the semantics side because their specifications. So to deal with that problem, we parse them anyway and then we process that unknown instructions by 2 different ways:

- **As skip instructions**: Continue exploring the program, knowing that the behavior of the skipped instructions is avoided and can cause undesired results. Can give false positives.
- **As stop instructions**: Avoiding all the behavior that is not defined, whenever we find an unknown instruction, the check done until that point will be totally correct, but future behavior won't be explored. Can give false negatives.

## 5.1 Paul Kocher examples

The Paul Kocher examples [10], are 15 different functions that exploit the SPECTRE v1 vulnerability by different techniques like loops or nested functions. The example number 10 is remarkable because its way to exploit the vulnerability: instead of using the loaded value from memory to index another access, it is used to verify a condition. This way, introduces a new way of exploiting the vulnerability, because to check what has been loaded in the first memory access, we are going to access always the same memory address.

### Results

#### Timing differences

A graphical representation of the running time difference of SPECTECTOR is given on Figure 5.1. It has been obtained by running SPECTECTOR with the version before this thesis was commenced and after it. The experiments used for this task have been the Paul Kocher examples [10], which don't implicate too complex performance on SPECTECTOR but they let us appreciate the differences when running it on simple examples.

We can clearly observe that in most of the cases the new SPECTECTOR version takes less time than the old one. The cases where it takes more, have been checked and correspond to examples not exhaustively inspected because the old version of tool didn't implemented the always-mispredict semantics 3.1.2.

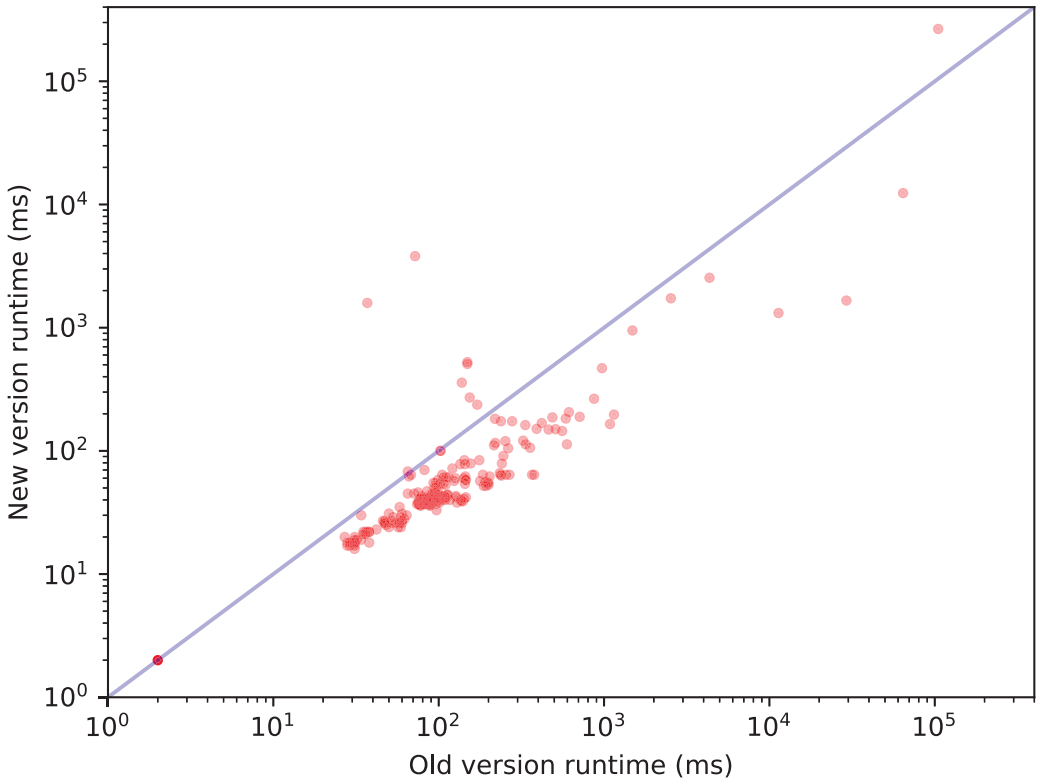



Figure 5.1: Timing SPECTECTOR over Paul Kocher examples
Each dot represents a concrete example, it is plotted with each of its coordinates representing the time taken to solve it. If a point is under the blue diagonal it means that it takes less time with the new version, otherwise, to be over the blue line means that takes more time with the new version.

On that cases, all the possible traces weren't explored, because when rollbacking, that action was propagated across all the states.

The changes observed are not only given by the modifications on the frontend, also on the analysis part. One of them, that has leveraged its performance is the usage of a main SMT process instead of launching a new one for each formula to solve, this way the performance can be improved.

### GCC testing

On Figure 5.2 we've tried SPECTECTOR over the Paul Kocher examples [10] like in [2] but with the GCC compiler mitigation proposal. If the mitigation is applied totally right, the two columns correspondent to cSLH should be filled completely with ●.

| Ex. | GCC | | | |
|---|---|---|---|---|
| | UNP | | cSLH | |
| | -O0 | -O2 | -O0 | -O2 |
| 1 | ○ | ○ | ● | ● |
| 2 | ○ | ○ | ● | ● |
| 3 | ○ | ○ | ● | ● |
| 4 | ○ | ○ | ● | ● |
| 5 | ○ | ○ | ● | ● |
| 6 | ○ | ● | ● | ● |
| 7 | ○ | ○ | ● | ● |
| 8 | ○ | ● | ● | ● |
| 9 | ○ | ○ | ● | ● |
| 10 | ○ | ○ | ● | ● |
| 11 | ○ | ○ | ○ | ● |
| 12 | ○ | ○ | ● | ● |
| 13 | ○ | ○ | ● | ● |
| 14 | ○ | ○ | ● | ● |
| 15 | ○ | ○ | ○ | ● |

Figure 5.2: Analysis of Paul Kocher examples compiled with GCC

For each of the 15 Paul Kocher examples [10], we analyzed the unpatched version (denoted by UNP) and the GCC mitigation proposal (denoted by cSLH). Programs have been compiled without optimizations (`-O0`) or with compiler optimizations (`-O2`) using the GCC compiler. ○ denotes that SPECTECTOR detects a speculative leak, whereas ● indicates that SPECTECTOR proves the program secure.

When compiling the examples with `-O0` optimization, it gives a leak on examples 11 and 15. On example 11, there's a control leak produced by the loop of the function `memcmp`. On example 15, although the first deference of `x` (`*x`) is masked, `x` will be dereferenced again to perform `array1[*x]` (so the previous masking won't have any effect).

With the other compilers, when analyzing example 10 compiled with UNP and `-O0`, it gave to us ○, in this case, with GCC, we get ●. To perform the if statement, `x` should be equal to `x & array_size_mask`. Then, inside the if statement it will be used the masked value instead the original one (`x`), this way, it cannot leak information because we know that masked value is going to be inbounds.

## 5.2 Custom examples

Some more examples based on Paul Kocher ones (a simple function that receives an input) have been introduced to avoid some compilers countermeasures effectiveness.

We're going to explain a few of the most remarkable examples, the rest can be found at the Appendix A. Then, on 5.2.1, we show a whole overview of running all the experiments with all compilers and their possible flags.

### 5.2.1 Remarkable experiments

Example 16 previously defined on 2.6, makes the GCC compiler to load with a masked index the first value, which will be always the same, but when doing the second memory access, it's content can be observed by the third memory access. This way, the observations can change between 2 different executions with the same initial configuration, what makes the mitigation insecure.

```
1 if (A[x] < size)
2   temp &= B[A[x] * 512];
```

Figure 5.3: SPECTRE vulnerable pattern 17 — C code

On Figure 5.3, there's a value retrieved from memory twice, first to do a comparison and then to access memory, but it's not allocated on a local variable (this can be optimized by the compiler). When using a high optimization level (`O2`), neither clang under SLH and ICC under FEN aren't able to detect the flaw, so the resultant program is flagged as vulnerable. VISUAL C++ isn't capable to detect and mitigate this vulnerable code neither.

```
1 if (C[x] < size) {
2   temp &= B[A[C[x]]];
```

Figure 5.4: SPECTRE vulnerable pattern 18 — C code

On Figure 5.4, we represent a combination of the previously presented examples 16 (Figure 2.6) and 17 (Figure 5.3). There are three memory access nested, but also, the value used for the comparison is the same than the obtained from the first memory access. VISUAL C++ isn't capable to detect and mitigate this vulnerable code neither.

```
1 void victim_function_v19(size_t x) {
2   if (x < size) {
3     temp &= B[A[x] * 512]; }}
4
5 int main(){
6   victim_function_v19(0);
7   victim_function_v19(42);
8   return 0;
9 }
```

Figure 5.5: SPECTRE vulnerable pattern 19 — C code

On Figure 5.5, the function is the most basic one, but we don't call it directly, we call a main function that afterwards calls it. Surprisingly, the resultant code using the second function as an entry point is vulnerable against VISUAL C++ & ICC compiled with with FEN and `O2`. It's also vulnerable the code produced by GCC & CLANG, both with SLH and `O2`.

On these optimizations, the call to the function will be always the same (is static because it's known at compile time). So the compiler doesn't seems to inject countermeasures against static code, only dynamic one.

```
1 uint8_t y = A[x];
2 if (x < size) {
3   temp &= B[y * 512]; }
4 else {
5   y = 0; }
```

Figure 5.6: SPECTRE vulnerable pattern 20 — C code

On Figure 5.6, the first memory access is done naively before the condition check, and in the case the condition is not verified, the information retrieved from memory is thrown away. This code compiled by VISUAL C++ & ICC (with FEN) and CLANG (with SLH) is vulnerable under the `o0` optimization. The value is loaded before the the condition check, so it's not hardened with the predicate state (or inserted the `lfence`). This way, the second memory access although is masked, it doesn't have any effect.

```
1 unsigned int y = 0;
2 for (; y < 2 && x < size; y++) {
3   x = A[x];
4   temp &= x; }
```

Figure 5.7: SPECTRE vulnerable pattern 21 — C code

On Figure 5.7, the memory access is done on a controlled loop of 2 steps. VISUAL C++ isn't capable to detect and mitigate this vulnerable code although it mitigates the similar $5^{th}$ example from Paul Kocher (as shown in [2]).

```
1 switch(*x){
2   case 0:
3   temp &= B[A[*x] * 512];
4   break;
5   default:
6   if (*x < size) {
7     temp &= B[A[*x] * 512];}}
```

Figure 5.8: SPECTRE vulnerable pattern 26 — C code

On Figure 5.8, the input value is differentiated, and by a switch expression, the condition for checking the in-bounds of `*x` is only evaluated in the case `*x` is not 0. This code is vulnerable in all the compilers, the switch condition is transformed in a set of branch conditions. Depending on the compiler, the check performed for line 3 (`*x == 0`) is not detected as a potential leak or the value `*x` is not hardened.

## Results

For the custom examples, we've performed the experiments on similar way to the ones performed on the original tool [2]:

| Ex. | VCC | | | | | | ICC | | | | CLANG | | | | | | GCC | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | UNP | | FEN 19.15 | | FEN 19.20 | | UNP | | FEN | | UNP | | FEN | | SLH | | UNP | | cSLH | |
| | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 |
| 16 | ○ | ○ | ● | ● | ● | ● | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● | ○ | ○ | ○ | ○ |
| 17 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ● | ● |
| 18 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ○ | ● |
| 19 | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ● | ○ |
| 20 | ○ | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ● | ● | ○ | ● | ○ | ○ | ● | ● |
| 21 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● | ○ | ○ | ● | ● |
| 22 | ○ | ● | ○ | ● | ● | ● | ○ | ● | ● | ● | ○ | ● | ● | ● | ● | ● | ○ | ● | ● | ● |
| 23 | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ● | ○ |
| 24 | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ● | ○ |
| 25 | ○ | ○ | ○ | ● | ○ | ● | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● | ○ | ○ | ● | ● |
| 26 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ● |
| 27 | ○ | ● | ○ | ● | ○ | ● | ○ | ● | ● | ● | ○ | ● | ● | ● | ● | ● | ○ | ● | ● | ● |

Figure 5.9: Analysis of custom examples

For each of the 12 custom examples, we analyzed the unpatched version (denoted by UNP), the version patched with speculation barriers (denoted by FEN), the version patched using speculative load hardening (denoted by SLH), and the GCC one (denoted by cSLH). Programs have been compiled without optimizations (`-O0`) or with compiler optimizations (`-O2`) using the compilers VISUAL C++ (two versions), ICC, CLANG, and GCC. ○ denotes that SPECTECTOR detects a speculative leak, whereas ● indicates that SPECTECTOR proves the program secure.

With these examples, we've demonstrated to find more cases not covered by compilers only by looking their behavior when inserting mitigations.

For ICC, it gives the possibility of mitigate SPECTRE variant 1 by inserting fences. On Paul Kocher examples [10] it didn't produced any vulnerable code (experiments on [2]), but we've demonstrated that on some cases, it can produce vulnerable code. As demonstrated, compilers optimizations let controlled values to not be hardened. It has been shown that the optimizations used over the compilation can affect on on the programs applied mitigations, so we cannot rely completely on how the compiler mitigations will behave.

## 5.3 XEN *hypervisor project*

For evaluating the capability of a static analysis tool such as SPECTECTOR over big projects, we've chosen the XEN *hypervisor project*, which *"is focused on advancing virtualization in a number of different commercial and open source applications, including server virtualization, infrastructure as a service (IAAS), desktop virtualization, security applications, embedded and hardware appliances, and automotive/aviation"* XEN *hypervisor project*.

So, to detect the flaws produced by SPECTRE v1 is so important because it hasn't been fixed yet and it's difficult to detect. If a SPECTRE v1 flaw is detected on software such as

this one that runs on critical systems, and most important can be exploited, it supposes an important security risk for the whole system and what it involves, letting a malicious user retrieving arbitrary data from memory.

The fact that a shared virtualization software has a vulnerability like this, makes it insecure, because it will be the software the one that manages all the users memory contents and if an user has sensitive data allocated, it will be retrieved by another user by only exploiting the vulnerability. Also, a software like this, which is used on the cloud, can be exploited against other users, what makes it more dangerous.

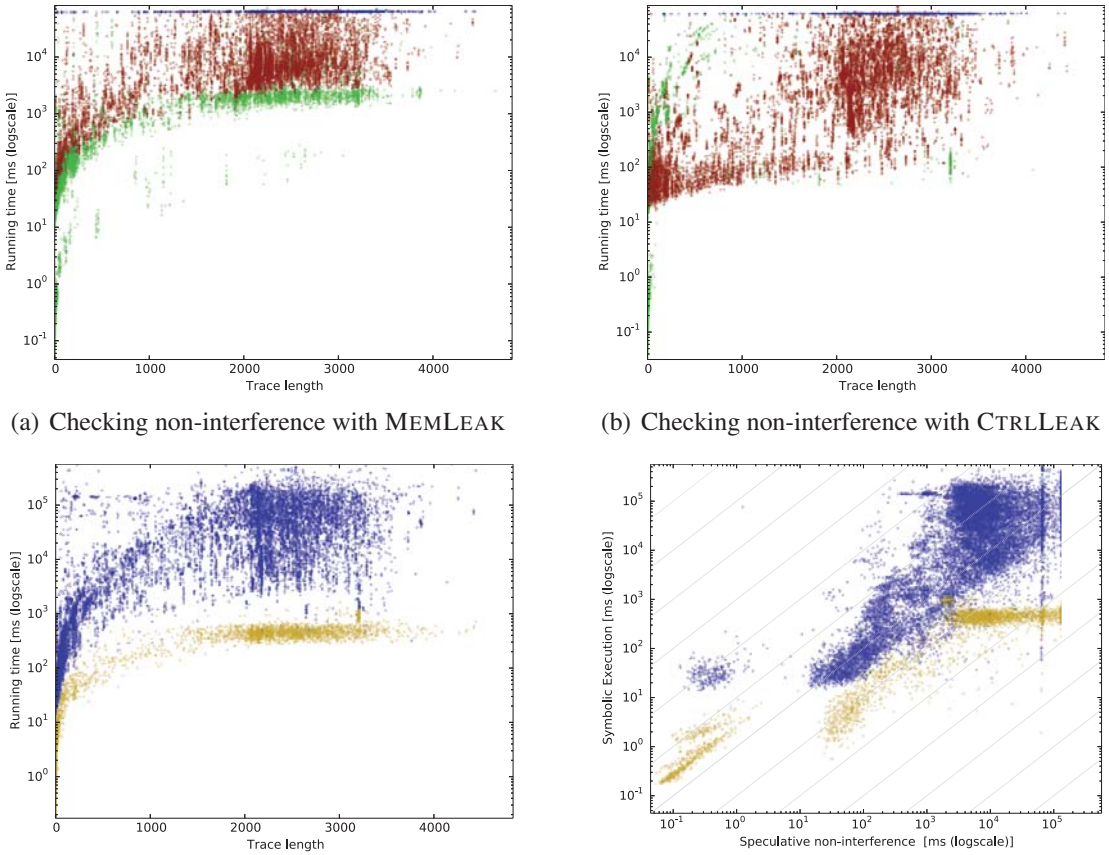
### 5.3.1 Results

On Figure 5.10, it's shown the interpretation of the statistics collected over the analysis of the whole XEN *hypervisor project*. The project has been linked into an assembly file following the procedures described in 4.2 (using the CLANG compiler), the number of total files used for this process is of 353 and a total of 2923 functions, from which 466 have been flagged as insecure against SPECTRE variant 1 by the XEN *hypervisor project* team. It's important to see how does the time of the analysis scales with the length of the traces.

This project code doesn't contains so complex executions with so much internal recursions. Due that it's an hypervisor, the functions it contains must be as simple as possible because it's software that is going to be called frequently.

The problem to solve can be huge, given the number of bifurcations $N$, the number of SMT formulas is going to be $2^N$. Although that complexity, we don't observe that limits on the timing statistics. The observed scalability is linear regarding to the trace length, so we can infer that the number SMT formulas to solve doesn't affect so much or they aren't as many as expected.

We've managed to compile the XEN *hypervisor project* project with the FEN and SLH mitigations the LLVM compiler provides by using the `llc` utility when doing the projects linkage previously defined on 4.2.1. When using FEN, the resulting file has 64423 `lfence` instructions, whereas the original one has only 52. When using SLH, the resulting file is `22M` long, whereas the original is `15M` long, this may be because all the additional instructions inserted for the predicate state tracing and masking of memory indexes. This shows the capability of the compilers to produce hardened code, so this version can be analyzed the same way that it has been analyzed the UNP one.




(a) Checking non-interference with MEMLEAK

(b) Checking non-interference with CTRLLEAK

(c) Symbolic execution engine

(d) Symbolic execution versus SNI check

Figure 5.10: Scalability analysis for the XEN *hypervisor project* with UNP
In (a) and (b), green denotes secure traces, red denotes insecure traces, and blue denotes traces producing timeouts. In (c) and (d), yellow denotes the first trace discovered for each function, while blue denotes all discovered further traces. The vertical lines in (d) represent traces where either MEMLEAK times out and CTRLLEAK succeed or both time out.

# 6 Conclusions

We introduce a new method for detecting the speculative information flows on the large code-bases. All the source code used for this project can be found referenced at SPECTECTOR project web page (`https://spectector.github.io/`). For extending SPECTECTOR's working domain, the process of development was divided into the next phases:

- Extending the parser to behave with the memory models specified on the program directives
- Instrumenting SPECTECTOR with new features such as the statistics collection for detecting the bottlenecks of the process
- Developing a method for analyzing big projects by linking the files according their dependencies

By that process, we've been able to analyze all the XEN *hypervisor project hypervisor project* [25], showing the potential of SPECTECTOR for analyzing large code-bases. Also, after analyzing the project, we demonstrate the scalability the tool presents with the statistics.

We can plot the data generated by these experiments, all the correlations of the analysis timing and the rest of properties of the experiments (i.e. length of the SMT traces or program sizes). The correlation between the axes shown on Figure 5.10 is linear, what means that although the number of SMT formulas to be solved is exponential regarding to the number of branch conditions, it doesn't affect to the analysis.

Also, basing our knowledge on the existing compiler mitigations, we've managed to create insecure programs that aren't detected by the compilers, demonstrating that the mitigations may not work in all the cases.

Knowing how the compiler mitigations work, the cases where they don't work are easy to find as shown in 5.2. An orthogonal work to this thesis is to find the cases not covered by the compilers on reliable ways and apply the detection of those patterns over other programs.

There have been proposals to stop SPECTRE variant 1 with new microarchitecture models [26], but the built processors are going to still be vulnerable. So it's necessary to have a reliable system for detecting the speculative information flows on the current systems.

## 6.1 Security and performance

The examples compiled with CLANG and using FEN never get an insecure program (as ○), but compiler inserts fences in all the cases, what reduces the performance of the programs. Speculative execution will perform right predictions in most of the cases, that's why it's a reliable system. With SPECTECTOR we're detecting only the leaks In the case when there's a misspredict ion. On a normal execution, the case a leak is produced, it means that a misspredict ion has been done, but that is not the common case.

So we're talking that when mitigating SPECTRE v1, we may degrade the performance of all the program just for a specific case. It's also necessary to consider if the possible leaked memory is such important to degrade the performance by inserting a mitigation. And most important, if the vulnerable function is accessible by a potential attacker (that may know the architecture of the code that's executed when sending inputs).

We've demonstrated that we cannot completely rely on the compilers for doing the work of mitigating SPECTRE v1. They either will perform unnecessary mitigations or won't mitigate at all SPECTRE v1. SPECTECTOR helps us to do the task of finding the vulnerable patterns or verify that the code is secure. This way, the developer can ensure the code correct behavior without unnecessary over-costs on the performance.

# Bibliography


[1] 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.

[2] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: principled detection of speculative information flows", *CoRR*, vol. abs/1812.08639, 2018. arXiv: 1812.08639. [Online]. Available: http://arxiv.org/abs/1812.08639.

[3] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy", *TPLP*, vol. 12, no. 1–2, pp. 219–252, 2012, http://arxiv.org/abs/1102.5497.

[4] M. Miller, *Mitigating speculative execution side channel hardware vulnerabilities*, https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018.

[5] *Chromium site isolation*. [Online]. Available: http://www.chromium.org/Home/chromium-security/site-isolation (visited on 04/11/2019).

[6] *Spectre v1 mitigations on linux kernel 4.17*. [Online]. Available: http://lkml.iu.edu/hypermail/linux/kernel/1805.1/04025.html (visited on 04/11/2019).

[7] A. Pardoe, *Spectre mitigations in msvc*, https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[8] Intel, *Using intel compilers to mitigate speculative execution side-channel issues*, https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues, 2018. (visited on 11/13/2018).

[9] *Rl336990 - [slh] introduce a new pass to do speculative load hardening to mitigate spectre variant #1 for x86*, https://reviews.llvm.org/rL336990, 2018.

[10] P. Kocher, *Spectre mitigations in Microsoft's C/C++ compiler*, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. (visited on 11/16/2018).

[11] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[12] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel (s)", in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2002, pp. 29–45.

[13] 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)*, 2014, pp. 719–732.

[14] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, *Netspectre: Read arbitrary memory over network*, 2018. arXiv: 1807.10535 [cs.CR].

[15] V. Kiriansky and C. Waldspurger, *Speculative buffer overflows: Attacks and defenses*, 2018. arXiv: 1807.03757 [cs.CR].

[16] J. Horn, *CVE-2018-3639 - speculative store bypass*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639, 2018. (visited on 11/12/2018).

[17] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention", *CoRR*, vol. abs/1903.01843, 2019. arXiv: 1903.01843. [Online]. Available: http://arxiv.org/abs/1903.01843.

[18] Intel, *Intel analysis of speculative execution side channels*, https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf, 2018.

[19] Richard Biener, *Spectre V1 diagnostic / mitigation – GCC mail archive*, https://gcc.gnu.org/ml/gcc/2018-12/msg00113.html.

[20] K. Sen, "Concolic testing", in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007, pp. 571–572.

[21] *Microsoft compiler calling convention*, https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019,

[22] *LLVM calling convention*, https://llvm.org/docs/LangRef.html#callingconv,

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, Palo Alto, California: IEEE Computer Society, 2004, pp. 75–, ISBN: 0-7695-2102-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673.

[24] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing in science & engineering*, vol. 9, no. 3, p. 90, 2007.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", in *ACM SIGOPS operating systems review*, ACM, vol. 37, 2003, pp. 164–177.

[26] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors", in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 974–987.

# APPENDIX A: CODE FROM CUSTOM EXAMPLES

```
uint8_t mem_leak(size_t x) {
  if (x < size) {
    return A[x];
  }
  return 0; }

void victim_function_v22(size_t x) {
  mem_leak(mem_leak(x)); }
```

Figure A.1: SPECTRE vulnerable pattern 22 — C code

```
void victim_function_v23(size_t x) {
  if (x < size) {
    temp &= B[A[x] * 512]; }}

void attacker_function(){
  victim_function_v23(42);
  return; }
```

Figure A.2: SPECTRE vulnerable pattern 23 — C code

```
void victim_function_v24(size_t x) {
  if (x < size) {
    temp &= B[A[x] * 512];}}

void another(int y){
  if(y)
  victim_function_v24(42);
  return; }
```

Figure A.3: SPECTRE vulnerable pattern 24 — C code

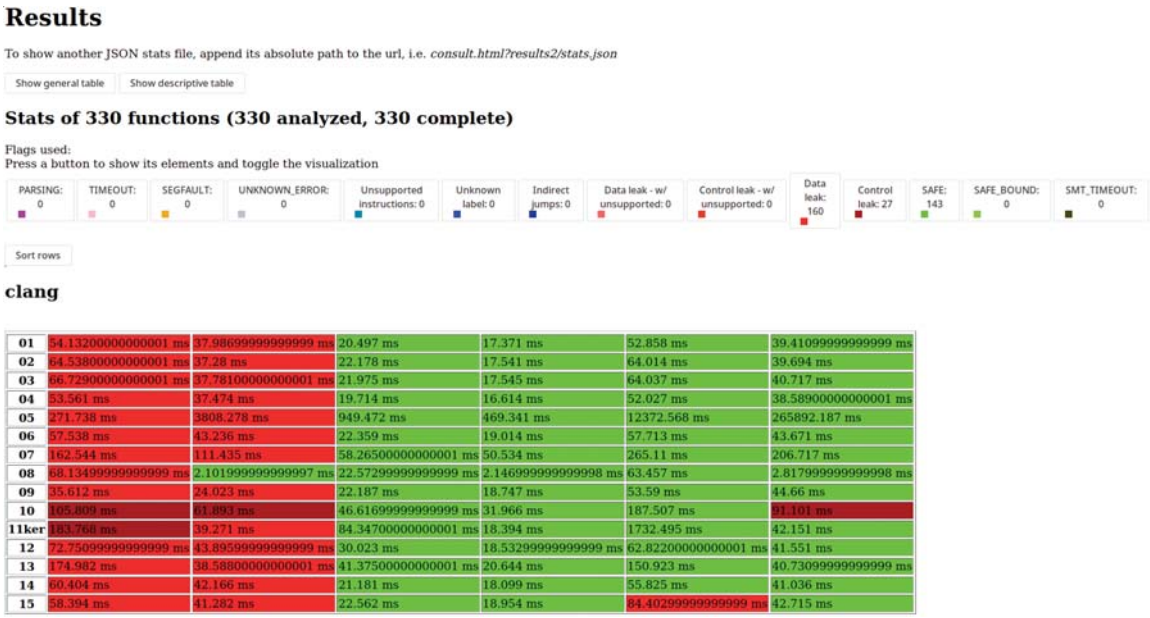
```
switch(x){
  case 0: temp &= B[A[x] * 512];
          break;
  default: if (x < size) {
           temp &= B[A[x] * 512]; }}
```

Figure A.4: SPECTRE vulnerable pattern 25 — C code

```
void victim_function_v27(size_t x, size_t y) {
  if (x >= size) {
    y = 0;
  }
  temp &= B[A[y] * 512]; }
```

Figure A.5: SPECTRE vulnerable pattern 27 — C code

# APPENDIX B: TABLES VISUALIZATION



**Results**

To show another JSON stats file, append its absolute path to the url, i.e. *consult.html?results2/stats.json*

Show general table | Show descriptive table

**Stats of 330 functions (330 analyzed, 330 complete)**

Flags used:
Press a button to show its elements and toggle the visualization

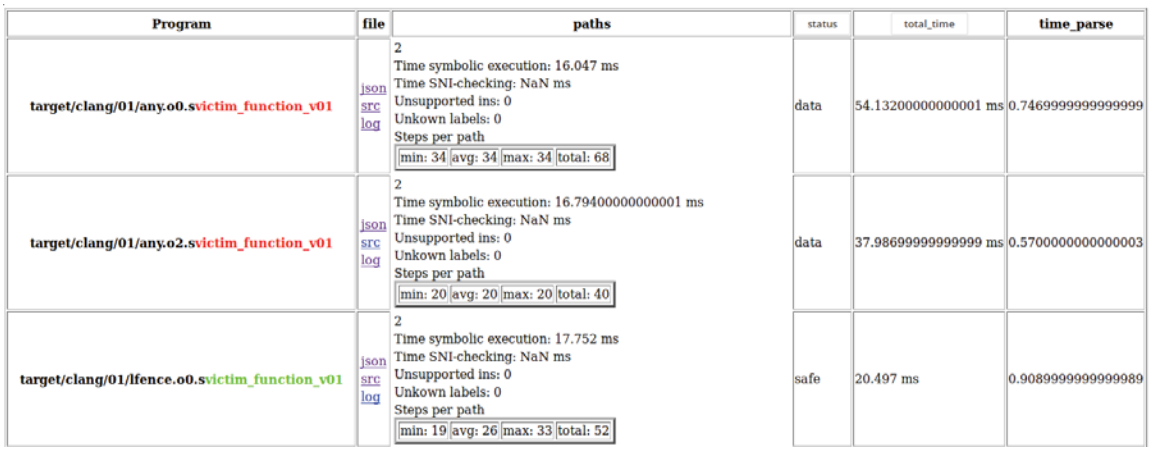
PARSING: 0 | TIMEOUT: 0 | SEGFAULT: 0 | UNKNOWN_ERROR: 0 | Unsupported instructions: 0 | Unknown label: 0 | Indirect jumps: 0 | Data leak - w/ unsupported: 0 | Control leak - w/ unsupported: 0 | Data leak: 160 | Control leak: 27 | SAFE: 143 | SAFE_BOUND: 0 | SMT_TIMEOUT: 0

Sort rows

**clang**

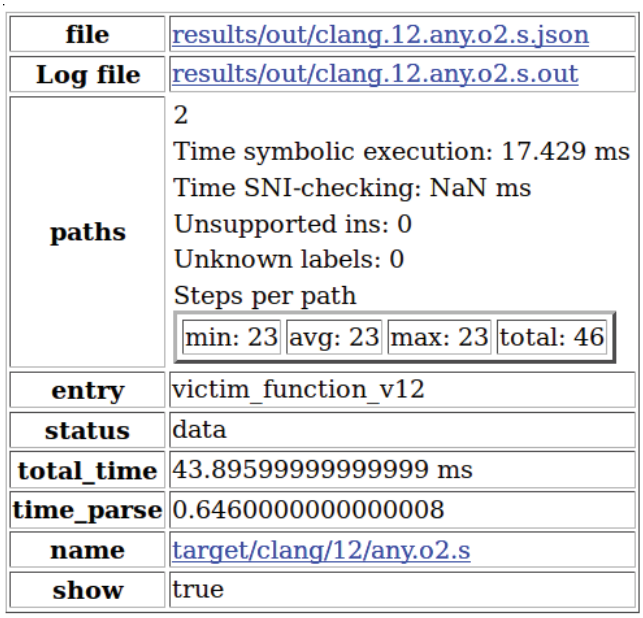
| | | | | | | |
|---|---|---|---|---|---|---|
| 01 | 54.13200000000001 ms | 37.98699999999999 ms | 20.497 ms | 17.371 ms | 52.858 ms | 39.41099999999999 ms |
| 02 | 64.53800000000001 ms | 37.28 ms | 22.178 ms | 17.541 ms | 64.014 ms | 39.694 ms |
| 03 | 66.72900000000001 ms | 37.78100000000001 ms | 21.975 ms | 17.545 ms | 64.037 ms | 40.717 ms |
| 04 | 53.561 ms | 37.474 ms | 19.714 ms | 16.614 ms | 52.027 ms | 38.58900000000001 ms |
| 05 | 271.738 ms | 3808.278 ms | 949.472 ms | 469.341 ms | 12372.568 ms | 265892.187 ms |
| 06 | 57.538 ms | 43.236 ms | 22.359 ms | 19.014 ms | 57.713 ms | 43.671 ms |
| 07 | 162.544 ms | 111.435 ms | 58.26500000000001 ms | 50.534 ms | 265.11 ms | 206.717 ms |
| 08 | 68.13499999999999 ms | 2.101999999999997 ms | 22.57299999999999 ms | 2.146999999999998 ms | 63.457 ms | 2.817999999999998 ms |
| 09 | 35.612 ms | 24.023 ms | 22.187 ms | 18.747 ms | 53.59 ms | 44.66 ms |
| 10 | 105.809 ms | 61.893 ms | 46.61699999999999 ms | 31.966 ms | 187.507 ms | 91.101 ms |
| 11ker | 183.766 ms | 39.271 ms | 84.34700000000001 ms | 18.394 ms | 1732.495 ms | 42.151 ms |
| 12 | 72.75099999999999 ms | 43.89599999999999 ms | 30.023 ms | 18.53299999999999 ms | 62.82200000000001 ms | 41.551 ms |
| 13 | 174.982 ms | 38.58800000000001 ms | 41.37500000000001 ms | 20.644 ms | 150.923 ms | 40.73099999999999 ms |
| 14 | 60.404 ms | 42.166 ms | 21.181 ms | 18.099 ms | 55.825 ms | 41.036 ms |
| 15 | 58.394 ms | 41.282 ms | 22.562 ms | 18.954 ms | 84.40299999999999 ms | 42.715 ms |

Figure B.1: Visualization result by tables



| Program | file | paths | status | total_time | time_parse |
|---|---|---|---|---|---|
| target/clang/01/any.o0.svictim_function_v01 | json src log | 2<br>Time symbolic execution: 16.047 ms<br>Time SNI-checking: NaN ms<br>Unsupported ins: 0<br>Unkown labels: 0<br>Steps per path<br>min: 34 avg: 34 max: 34 total: 68 | data | 54.13200000000001 ms | 0.7469999999999999 |
| target/clang/01/any.o2.svictim_function_v01 | json src log | 2<br>Time symbolic execution: 16.79400000000001 ms<br>Time SNI-checking: NaN ms<br>Unsupported ins: 0<br>Unkown labels: 0<br>Steps per path<br>min: 20 avg: 20 max: 20 total: 40 | data | 37.98699999999999 ms | 0.5700000000000003 |
| target/clang/01/lfence.o0.svictim_function_v01 | json src log | 2<br>Time symbolic execution: 17.752 ms<br>Time SNI-checking: NaN ms<br>Unsupported ins: 0<br>Unkown labels: 0<br>Steps per path<br>min: 19 avg: 26 max: 33 total: 52 | safe | 20.497 ms | 0.9089999999999989 |

Figure B.2: Descriptive table visualization



| | |
|---|---|
| **file** | results/out/clang.12.any.o2.s.json |
| **Log file** | results/out/clang.12.any.o2.s.out |
| **paths** | 2<br>Time symbolic execution: 17.429 ms<br>Time SNI-checking: NaN ms<br>Unsupported ins: 0<br>Unknown labels: 0<br>Steps per path<br>min: 23 avg: 23 max: 23 total: 46 |
| **entry** | victim_function_v12 |
| **status** | data |
| **total_time** | 43.89599999999999 ms |
| **time_parse** | 0.6460000000000008 |
| **name** | target/clang/12/any.o2.s |
| **show** | true |

Figure B.3: Table about an specific example
Representation of Paul Kocher example 12 compiled with CLANG, using the options UNP and -02

Este documento esta firmado por


| | |
|---|---|
| **Firmante** | CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES |
| **Fecha/Hora** | Sat Jun 08 22:52:03 CEST 2019 |
| **Emisor del Certificado** | EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES |
| **Numero de Serie** | 630 |
| **Metodo** | urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature) |