Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inspectre Gadget: Discovery of Speculative Disclosure Gadgets in the Linux Kernel

Bachelor Thesis

Author: Loris Sikora

Tutor: Johannes Wikner

Supervisor: Prof. Dr. Kaveh Razavi

February 2022 to July 2022

Abstract

A few years ago, a hardware vulnerability using speculative execution attacks that exploit branch target injection was uncovered, which allowed processes to read from higher privileged memory. In order to exploit this weakness, we require small code snippets, referred to as disclosure gadgets. These are to be found within existing binaries and are the key to extracting values from memory we, as the attacker, normally should not have access to. Finding these gadgets manually is completely infeasible, due to the size of most binaries. In this thesis we present *Inspector Gadget*, a scanner which, given a set of constraints, can locate these gadgets within an ELF binary using symbolic execution. Ultimately, we were able to find a second, previously unknown speculative disclosure gadget.

Contents

Chapter 1

Introduction

A few years back, the world was introduced to the concept of speculative execution attacks[7, 8], some of which are performed by using branch target injection (BTI). To perform these attacks and extract memory from a privileged process, we require small code snippets, referred to as "Speculative Disclosure Gadgets", that are located in the kernel. Several software mitigations[6, 2] have been deployed in order to patch these hardware vulnerabilities. It has, however, been recently shown[15] that some of these mitigations can be bypassed under specific circumstances. As such, the need for such gadgets is still prevelant, in order to develop end-to-end exploits using Spectre-BTI. *Inspectre Gadget* is here to serve this need by providing a tool which locates speculative disclosure gadgets.

1.1 Motivation and Challenges

One of the key motivators for this thesis was the paper "Retbleed Arbitrary Speculative Code Execution with Return Instructions"[15]. It shows how mitigations against the Spectre vulnerability can still be circumvented. This is done by showing how return instructions can behave like indirect branches, and how an unprivileged attacker can arbitrarily control the predicted target of such return instructions. The jump target of such an attack would be a speculative disclosure gadgets, one of the focal points of this thesis.

As research continues and more possibilities to execute Spectre-BTI will inadvertantly come to light, the need for disclosure gadgets will continue to exist as it is common to provide some form of proof of concept (PoC). The authors of Retbleed developed a gadget scanner in order to find a fitting gadget to demonstrate the newly discovered circumvention. *Inspectre Gadget*, the product of this thesis, aims to be a more advanced and in-depth gadget scanner. Using symbolic execution and some custom analysis techniques, *Inspectre Gadget* is able to find more gadgets than in previous work.

All gadgets follow a similar structure. They may, however, slightly vary from it, which is the main challenge of this task. Using symbolic execution together with additional simplification and analysis on generated abstract syntax trees of values, we intend to account for these deviations and allow for the discovery of even more complex gadgets. We will be able to specify a detailed initial state where we can declare constraints through three main means of control description: Controllable register values, constant register values and controllable memory.

Additionally, some ISAs can be quite complex and provide a plethora of different possibilities to achieve the same goal. The scanner needs to be able to handle this. This is ought to be done by utilizing libraries such as *Capstone*[14] and *angr*[13] which provide support for many ISAs. *angr*, a library used for symbolic execution, will bring some limitations with it, which need to be overcome first.

1.2 Overview

Chapter 2 goes over what the Spectre vulnerability is and how it works. It lays the foundation about speculative execution, details what disclosure gadgets are and how they can be used to extract memory.

Chapters 3 and 4 will present the design and final implementation of the gadget scanner, the main focus of this thesis.

Finally, chapter 5, 6 and 7 show the result of an analysis run on a recent Linux kernel, compare these results to other existing tools and reflect over the project as a whole.

Chapter 2

Background

2.1 FLUSH+RELOAD attack

Spectre attacks perform a `FLUSH+RELOAD`[16] attack among other things. Normal operations don't allow for direct reading of caches. What is possible, however, is that a process A might infer certain information about a different process B, assuming they share memory, via the cache. Process A can do so by flushing the shared memory region from the cache and waiting for B to perform an access. Once this has occured, A will load the entirety of the region and can determine, based on how long the access takes, whether a value was already in the cache or not, and can thus roughly tell where the load has occurred. This is commonly referred to as a `FLUSH+RELOAD` attack (a variant of `PRIME+PROBE`[11]) and is a form of side-channel attack, where the side-channel in this case is the cache.

It is important to note that an attacker won't be able to detect this change for a single byte. Since caches are commonly organized in pages, we will only be able to determine the cache line within which the load occurred. But as we will see in subsequent chapters, this can already be enough to leak information.

2.2 Speculative execution

In order to improve performance, modern processors have been equipped with a feature called speculative execution. Due to the fact that certain operations can take many CPU cycles to perform, it can happen that execution may stall at a control flow edge. Using branch prediction, the process of performing an educated guess over whether a branch will be taken or not based on previous outcomes, the processor will speculate and preemptively execute the branch that is considered more likely. Similarily, for always-taken indirect branches, the branch predictor needs to predict the branch target. At some point, the often time-consuming operation will have been completed, and we will be able to tell for certain whether the branch should have been taken or not. Should the prediction prove to be correct, we just saved a great amount of time, as we have already pre-computed a potentially big portion of the correct branch. Should the guess turn out to be wrong, however, the system will be forced to roll back any changes made to registers or memory.

2.3 Spectre Attack

In 2018 a paper titled "Spectre Attacks: Exploiting Speculative Execution" [7] was released. The paper describes a technique for exploiting speculative execution in processors to extract values from anywhere within a program's memory space via side-channels. Two different variants, called Spectre v1 (bounds check bypass) and v2 (branch target injection – BTI), were presented, the latter of which will be the focus of this thesis.

Modern operating systems provide each process with some reserved memory that in theory only they have access to. In addition, there may be privileged memory that can only be accessed by the kernel. Should a normal user program try to access such privileged memory, the CPU will raise a fault. However, if a program tries to read values that it isn't supposed to while the processor is speculatively executing, the system will not yet be aware of the violation, as faults are temporarily supressed during speculative execution. If a branch prediction turns out to be correct and the executed code contains an illegal instruction, the CPU will be able to detect this at a later point and will handle it accordingly. Was the prediction wrong, the rollback will occur, and all violations will be ignored. The issue that was presented as the Spectre attack, is that while the rollback will clear any illegally read values from registers and memory, they still might be accessible through a side-channel. One of which is the cache residing between the CPU and the main memory.

To perform an attack, the main goal is to enter speculative execution. In order to do this, we need to step over a branch multiple times to "train" the branch predictor to predict a location, which we will refer to as the gadget. This branch can be an indirect branch, as shown in the case of Spectre v2, or a return instruction for the technique described in RETBLEED. After we have established this prediction behavior, we unexpectedly change the destination and thus cause the gadget to be speculatively executed. It is now very important what the gadget does, which leads us to the next step.

The gadget has to perform a `FLUSH+RELOAD` attack for us: First, it must read a small portion of an illegal value (called the `*secret` from now on) and use it as an offset to dereference an array called the reload buffer (`*rb`), which is allocated by us and located in shared memory. Concretely, such a dereference could look like this:

```c
y = rb[secret[x] * 4096]
```

Listing 1: Example Spectre gadget in C

```asm
movzx %eax, byte ptr [%rax];
shl %rax, $12;
mov %r9, qword ptr [%rax+%r8];
```

Listing 2: Example Spectre gadget in Assembly (`%rax = *secret` and `%r8 = *rb`)

Depending on the secret byte, a different area of our reload buffer will be loaded. It is important to note that the value loaded with the second dereference does not matter at all. The processes of dereferencing is what counts. This also implies that the last operation doesn't necessarily need to be a load, it might just as well be a store operation that causes the dereferencing to happen.

The factor 4096, which gets multiplied with the secret, is here to "stretch" each possible value of the secret to a different cache page, which, as already hinted towards in section 2.1, greatly improves our ability to extract the secret. This is because we will now be able to explicitly tell what the secret byte was prior to dereferencing, as determining the warmed up page will now uniquely indicate what the value of the byte was. Though technically, not absolutely necessary, the multiplication factor will greatly increase speed, as we can determine a byte with a single `FLUSH+RELOAD`. Should we not get this luxury, we can perform an initial `FLUSH+RELOAD` and continuously offset the secret byte (using `%r8` in the example above) and perform a subsequent attempt until we detect a different page being loaded. This indicates that we have hit a page boundary and the value of our secret byte is the start address from the following page minus the offset we have just established. The conditions for the `FLUSH+RELOAD` attack here are ideal, as we, in a sense, control both the attacker and the victim and don't need to wait for the latter to perform their action.

This small code snippet we jump to is called a speculative disclosure gadget, and is able to (generally) give us a single byte of desired memory, allowing us to read the complete secret byte by byte. Since the disclosure gadget is responsible for reading the secret, it will be required to be located in the same security domain as the secret and can thus not be written by the attacker themselves. As shown in recent work [15], an unprivileged attacker can poison kernel branches. But due to memory protection [2] the injected branch target must reside in kernel memory. This means that in order to leak memory from the kernel using BTI, we must find gadgets inside the kernel memory.

We now understand why these gadgets are important, what they are used for and most importantly that they must lie within instruction memory beyond our controllability, thus the need for us to find them in existing binaries. Depending on the execution state when the speculative control flow is hijacked, we have different control over registers and memory related to these registers. This means that we might require different gadgets accordingly. But how would we find a gadget that performs a `FLUSH+RELOAD` attack for us within the victim's address space?

Chapter 3

Design

Our task is to find disclosure gadgets, that are located within a target, like for instance the Linux kernel and its kernel modules. This can be done by sweeping through all executable sections of the roughly 45 MB binary byte by byte and checking whether the instructions at any of the bytes form a gadget. Discovering gadgets, such as the one shown in Listing 2, wouldn't be too difficult. But what for instance the findings of RETBLEED have shown us, is that their presented technique allows for only very few values of our machine (such as registers or memory contents) to be controlled. This requires for us to add a flexible constraint system with which we can describe the controllability of the machine state (discussed further in section 4.1) in order to filter out unsuitable gadgets.

In the end, we will have a scanner that takes in a binary and some constraints and finds us as many gadgets conforming to the given constraints as possible.

3.1 A first attempt

Initially, the idea was to write the gadget scanner solely using *Capstone*[14], a library that specializes in disassembly of programs. Given a binary or a string of bytes, it will decode them into their respective assembly instructions. It supports a variety of hardware architectures and provides bindings to its backend, which is written in C, through many languages. Capstone simplifies the task quite a bit, but doing the analysis on assembly instructions has proven to be still quite difficult. Not only would the gadget scanner need to perform the difficult task of handling large instruction sets such as x86 but, in addition to that, also support multiple ISAs similar in size. Former being especially challenging, as x86 has a lot of different ways to perform memory loads and stores (different sizes, multiple ways of addressing the location, etc.), which is core to the task we are trying to solve. A similar approach was used in RETBLEED [15].

The next idea was to use a lifter to elevate the assembly instructions to an intermediate representation, on which the analysis then could be performed. In this context, however, the idea of symbolic execution came to mind.

3.2 Symbolic execution

The second attempt was done using symbolic execution. Symbolic execution, at its core, is very similar to regular execution. The main difference being that it allows for values to be non-concrete. Register contents and memory values may be multiple constants at a time, contain symbolic placeholders or, most importantly, abstract syntax trees (ASTs) of the two. ASTs are expressions composed of constants, symbols and operations on them. Using ASTs, you cannot only describe the value itself but also what operation led to said contents. This proves to be particularly helpful if you want to solve for a variable within the tree, given a set of constraints. For instance, if you want to know the input value leading to a desired output.

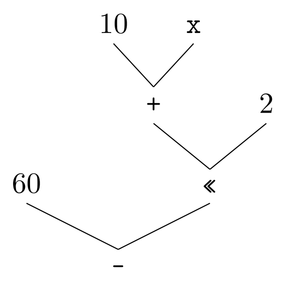


Figure 3.1: Example AST

Such an AST, as displayed in Figure 3.1, is ideal for our use case as we can fill registers and memory with placeholders, symbolically execute the program and check whether ASTs of the addresses used during loads and stores resemble the shape of a gadget.

3.3 *angr*

"*angr* is an open-source binary analysis platform for Python. It combines both static and dynamic symbolic ("concolic") analysis, providing tools to solve a variety of tasks."
– *angr.io[13] (July 22)*

As already implied by the quote above, *angr* is a Swiss army knife when it comes to binary analysis. It provides many useful tools for disassembly, lifting, decompilation, control flow analysis, support for most major CPU architectures and, most importantly for us, symbolic execution. It does so by combining many open-source projects to create one coherent library. From simple analysis and reverse engineering to vulnerability discovery and exploitation, *angr* can be used in a variety of fields.

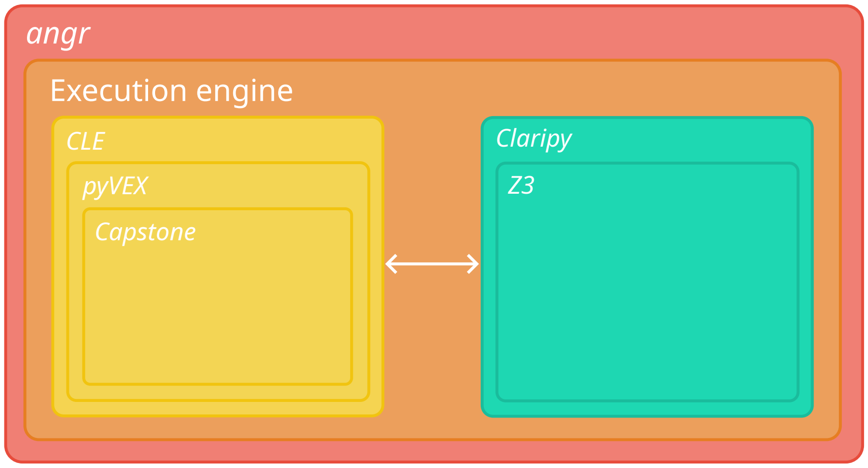


Figure 3.2: Overview of *angr*'s different components

For our cause, we require a few crucial things. These include a binary loader and disassembler to have access to instructions at a specific address, a lifter to some IR to allow for more detailed analysis and most importantly a symbolic execution engine that produces ASTs for registers and memory values for the detection of gadgets. Thankfully, *angr* provides all of that:

CLE[13] The main binary loader of *angr*. It uses *Capstone*[14], a library for disassembly. *Capstone* itself already provides a great benefit, as it supports a variety of architectures.

pyVEX[13] *angr* uses a mirror of *Valgrind*'s[9] IR language called *VEX*. *Valgrind* is an instrumentation framework developed for building dynamic analysis tools. In its own words, it can be used to analyze large applications and even automatically detect memory management errors and threading bugs. The decoded instructions from *Capstone* are forwarded and then lifted to *VEX*. This provides *angr* with greater flexibility for later analysis.

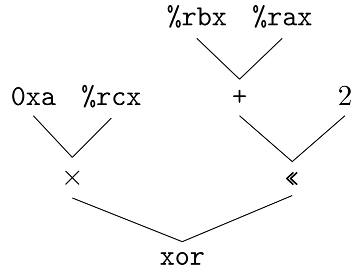
Claripy[13] is *angr*'s self-developed solver wrapper. It provides mainly an abstracted interface for constraint solvers. Its default backend is *Z3*[1], a powerful theorem prover developed at Microsoft Research. Using *claripy*, it is possible to describe values as ASTs. It allows for detailed analysis of assembly, such as finding an initial state to reach certain branches.

As such, when combining all mentioned tools, running an analysis on the assembly instructions

```
add %rax, %rbx;
shl %rax, $2;
mul %rcx, $0xa;
xor %rax, %rcx;
```

Listing 3: Assembly of symbolic execution example

will produce the following AST in register `%rax`:



`%rax = ((%rax + %rbx) << 2) xor (%rcx * 0xa)`

Figure 3.3: Symbolic execution example

angr is extremely modular, which albeit can be quite complex and confusing at times, allows for near full control over the library, by replacing specific parts with your own. Exchanging components or modifying the functionality of certain elements can be either done via *angr*'s hooking system or by extending and overriding a class within its main object responsible for simulation, the engine.

There are still some hurdles which need to be overcome first in order to take full advantage of *angr*'s capabilities. This process is described in greater detail in section 4.2 *angr* modifications.

We now have the resources to produce these ASTs. Having talked about some parts and design of such a hypothetical scanner, how would we actually connect them and use them to find gadgets? The next steps will involve passing them through the following three stages of *Inspectre Gadget*, simplification, control analysis and detection. As their names suggest, they will take ASTs and simplify, analyze which parts of the tree are controllable by the attacker and inspect them to determine if their shape resembles a gadget or not, all based on a set of constraints we provide.

This leads us into chapter 4, where we will go into detail about the actual implementation, necessary modifications to *angr* and techniques to detect gadgets.

Chapter 4

Implementation

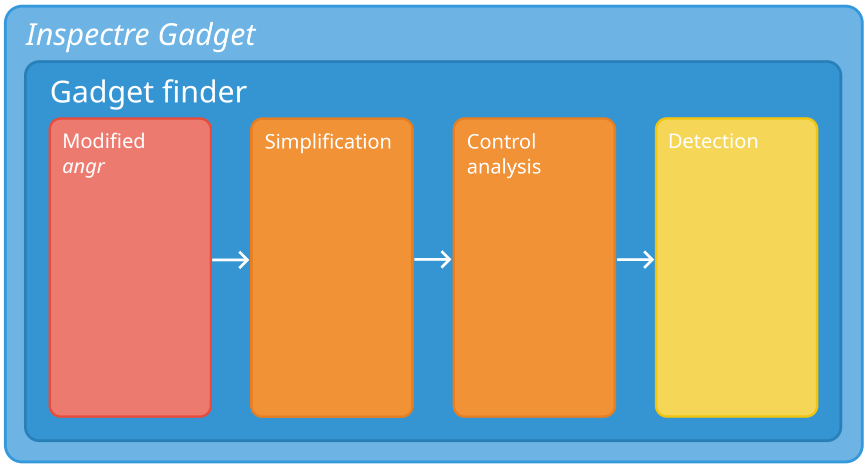


Figure 4.1: Overview of *Inspectre Gadgets* different components

Our gadget scanner can be divided into sevral parts. To detect whether a section contains a gadget, the driver iterates over all possible starting addresses. For each address, the block of assembly is symbolically executed using *angr*. Every time we encounter a read or write to memory during execution, we analyze the aforementioned AST of the address. This AST is passed through the simplification, control analysis and detection steps to determine whether it's a gadget or not. The following sections provide a more in-depth description of these main components.

4.1 Constraints

Assuming we have full control over all states of registers and memory, it would be easy to find thousands of gadgets within some applications, simply due to their size. However, we don't necessarily have full control over what is happening within the machine at the point of exploitation. Constraints allow for a detailed description of what we, as the exploiter, have direct control over or knowledge of.

Controllable registers This is a list of all registers we can fully control. In a sense, this is one of the most desirable situations, as it allows for near total freedom when describing the reload buffer and offset from the secret. Inside *angr*, these are implemented using symbolic BitVectors (BVS), which are stored in their corresponding registers.

Constant registers If a register is not completely controllable, we still might know its value. This can be extremely helpful for the gadget scanner while simplifying, as it will be able to resolve register reads while symbolically executing.

Controllable memory Using this list of memory ranges, we can tell the gadget scanner which regions of memory we have full control over. An address that can be resolved into one of these ranges will be turned into what is referred to as a symbolic memory AST, a BVS similar – although slightly more specific – to the ones used in controllable registers.

Displacements Displacements are slightly more specific memory references composed of a base address stored in a register combined with a constant offset, e.g. `0x10(\%r14)`. They are essentially syntactic sugar for controllable memory, which is also why, internally, they are implemented using constant registers and controllable memory.

4.2 *angr* modifications

As mentioned before, *angr* is very modular and therefore quite extensible. *angr* uses a component referred to as an "engine" as its mean of simulation. It's a class composed of many mixins that each handle different functions such as memory, concretization, syscalls, etc. By default, memory addresses get concretized once they're used in a load or store. Should this process fail in the case of a load, *angr* will simply return an `UNINITIALIZED` memory symbol. This is not ideal, as we would like to retain knowledge of multiple derefs inside our AST and not have them reduced to a single symbol. To cope with this, we create our own custom `angr.engines.vex.heavy.SimStateStorageMixin`, which is responsible for handling memory operations within *angr*, overriding the load and store handlers and inserting some logic. By introducing a custom AST type called `Deref`

```python
class Deref(claripy.ast.BV):
    def __repr__(self):
        return "mem[%s]" % (self.args[0])
```

Listing 4: `Deref` AST type used for wrapping dereferenced addresses (`engine.py:7`)

we prevent concretization from happening by wrapping the symbolic address with a Deref:

```python
class CustomSimStateStorageMixin(SimStateStorageMixin):
    def _perform_vex_expr_Load(self, addr, ty, ...):
        if addr.symbolic: # Deref
            return Deref(addr)
        else: # Default behaviour
            return self.state.memory.load(
                addr,
                self._ty_to_bytes(ty),
                ...
            )
```

Listing 5: Simplified logic to handle derefs (`engine.py:12`)

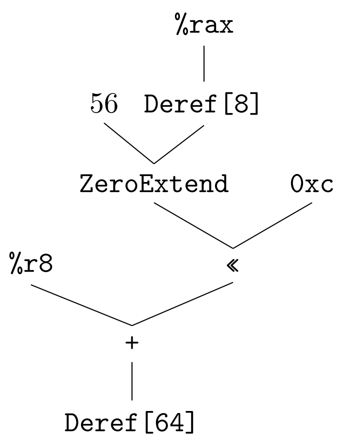
We can now retrace the source of our memory load at a later point in time by looking at the still preserved symbolic address contained within the Deref object. Unfortunatly, it wasn't this easy. The whole process involved considerable tweaking of *angr*'s more than 130 options in order to get it to behave in the desired way. One of the main challenges was to prevent the whole system of trying to concretize ASTs containing our derefs, which is done as a measure of optimization, as this would result in a crash.

With all of this in place, we can now symbolically execute assembly instructions, and *angr* will provide us with ASTs for each register. The following gadget,

```
movzx %eax, byte ptr [%rax];
shl %rax, $12;
mov %r9, qword ptr [%rax+%r8];
```

Listing 6: Example gadget

for which our final dereferenced reload buffer would be stored `%r9`, would produce the following AST:



```
%r9 = mem[ ((0#56 .. mem[%rax]) << 0xc) + %r8 ]
```

Figure 4.2: Graph representation of the AST of reg `%r9` from the simple gadget

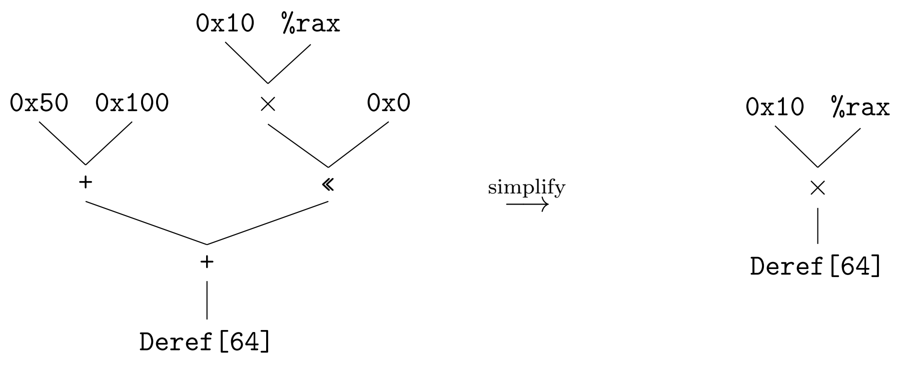
4.3 Simplification

In order to make the detection of gadgets easier, we first simplify the AST. Once a load or store has been reached, the corresponding AST of the address will be passed to this step. Many operations, although almost certainly influential on the result, can be neglected during the gadget analysis. One example for this would be the `add` operation. The addition of a fully controllable register and a constant still results in a value that we fully control. The reason for this is that, for any constant, we can simply counteract the offset by subtracting it from the register ahead of time to achieve the desired value. As such, offsets caused by addition and subtraction can be ignored.

The same goes for operations that we know (e.g. using our constraints) will have no effect (e.g. multiply by 1, add with 0). These, although commonly not really found in modern assembly, can still occur whenever we perform things like constant folding or register resolution. The simplification step can be quite impactful, as it can directly influence whether an AST might be too complex for the later detection phase or not.

Following is a visualization of the simplification process applied to an example:

Figure 4.3: Example before/after illustrating simplification



4.4 Control analysis

Sometimes we cannot simplify terms enough to make it easy for the detection phase to recognize certain parts of the gadget as controllable. We therefore added a system called *control analysis*, which, in addition to simplification, keeps track of controllability of nodes within the AST. It is essentially an additional, identically formed graph on top of the already existing tree. Each node stores the type of controllability for the resulting value as well as which registers and memory regions are required to maintain said controllability. One such example of an invalid gadget would look as follows:

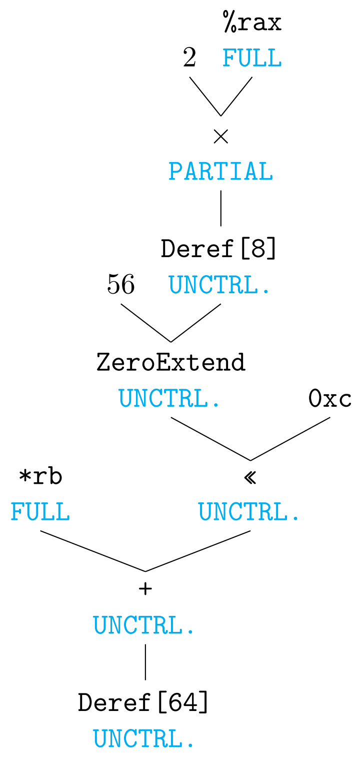
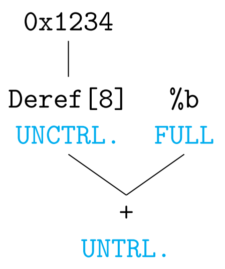


Figure 4.4: Control analysis of an invalid gadget

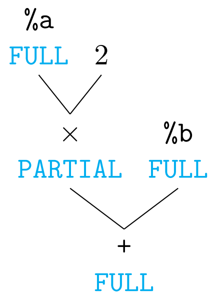
There are three types of controllability:

Type	**Description**	**Example**
`UNCONTROLLABLE`	The value is completely uncontrollable and unpredictable.	An arbitrary memory read.
`PARTIAL`	Some bits of the value are controllable but not all of them.	The register `%rax` after we've declared `%eax` as fully controllable.
`FULL`	The value is fully controllable.	A fully controllable register.

It is important to make the distinction between partial and complete lack of control. This difference is illustrated by the two examples in Figure 4.5.



(a) Uncontrollable example



(b) Controllable example

Figure 4.5: Example illustrating control analysis types and their impact

In the first example, we can see that due to the memory read being completely uncontrollable, adding it to `%b` will not increase controllability in any way. The resulting value will be completely uncontrollable.

For the second example, the product `%a * 2` will be classified as partially controllable, as we can only describe the even values from the whole range of possible numbers and thus lack control over the least significant bit. But since we then add it to register `%b` we can compensate for the evenness. Or, even simpler, we can set `%a` to `0x0` and only use `%b`, making the final value completely controllable again. This obviously requires the two registers `%a` and `%b` to not be the same.

In retrospect, these values reflect more on the state of predictability rather than controllability, as it has proven to be useful to consider constants as "fully controllable" in some cases.

The type of controllability is only half the story. As demonstrated above in Figure 4.5, it is significant to know the correlation (in terms of used registers and memory regions) between two branches of a tree. For each operation, the program considers the required registers and memory regions of all operators and tries to merge them. Re-using the second example shown above, if we have a multiplication using the fully controllable register `%a` we will propagate `%a` as a required reg to control the partially controllable result `%a * 2`. We can now use this information while processing the addition. The second operator is fully controllable, requiring reg `%b`. Should the two sets of required registers intersect, we will return `PARTIAL` controllability and use their union as required registers for our addition node. Otherwise, we return `FULL` controllability and simply forward the requirements from the fully controllable operator. It is important to note that this analysis is designed to make conservative estimates an tries to decrease the likelihood of false positives.

This kind of analysis now provides us with the ideal conditions to easily check if a secret pointer in a gadget can truly be used as such and if it in any way interferes with the reload buffer pointer.

4.5 Detection

Thanks to the previous steps, the detection process is now quite straightforward. The detector is responsible for checking whether the AST of an address used in a store or load is a gadget. This is done by walking through the AST and checking for key features such as shape and controllability. For instance, as already touched upon in section 2.3, the dereferenced secret of a gadget doesn't necessarily need to be shifted.

Something else that might happen is that the offset, which is added to the secret byte (`%r8` in Figure 4.2), isn't necessarily a register. It might also be a fully controllable value composed of operations containing controllable memory and regsiters. Additionally, there are multiple ways of expressing the dereferencing of a single byte inside an AST. For instance, the byte pointer could be dereferenced and the resulting value zero-extended to 64 bits (`ZeroExtend(56, Deref[8](*x))`), or the pointer might be dereferenced into a 8bit register, where the upper part is taken from the previous value of the full sized counterpart (`Concat(%a[0:56], Deref[8](*x))`). For the second case we will have to ensure that the upper part of the full-sized register is under full control.

As you can see, there are still quite some variabilities regarding the shape of a gadget, but in general, the desired AST shape and control of a gadget is as presented in Figure 4.6.

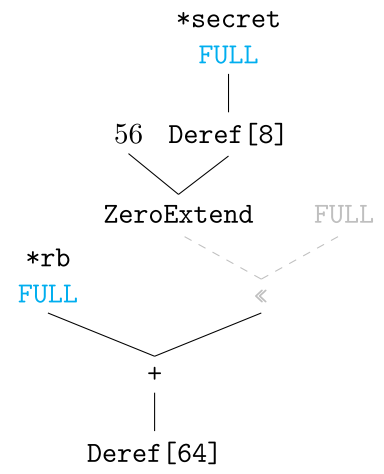


Figure 4.6: AST shape of a gadget with optional shifting and `*rb` and `*secret` inserted as fully controllable placeholders, that don't interfere

4.6 Parallelization

Initially, the program was designed to run through a range of addresses sequentially. This quickly became infeasible as development progressed, as the expected runtime for the Linux kernel would be multiple days to weeks. Since the task is very repetitive and the different trials don't depend on each other, the problem is ideal for parallelization.

Splitting the workload equally amongs a specified amount of threads revealed the inherit flaw of *angr* not being thread-safe. A subsequent attempt using python's `Process` class from the `multiprocessing` package was attempted. Forking and thus creating multiple, disjoint instances of *angr*, worked perfectly. When parallelization is enabled, the main process spawns a specified amount of subprocesses and assigns each one an equal amount of work. Every subprocess periodically reports its progress and any found gadgets back to the main process.

Due to the long runtime of the simulations and the ubiquitous risk of crashes, an intermediate save state system was put it place. Rougly, every minute, the complete state of progress will be saved to disk, from which can be recovered at any point in time. This save state tracks the progress of each process, as well as the gadgets that have been found up until that point. A fortunate side effect of this is, that it is possible to easily extract already found gadgets while the analysis is still running, not needing to wait for it to complete.

Chapter 5

Evaluation

5.1 Unit tests

To test the scanner's accuracy, unit tests were written that contained gadgets and code snippets similar to but in fact not actual gadgets. These tests were a mixture of custom tests and a set of tests originally written for the Retbleed scanner. The Retbleed scanner is unable to pass all of their unit tests, in contrast to *Inspectre Gadget*, which not only passes all tests but was also able to identify a false positive within the set of tests. We were able to use these tests early on as proof of concept for the different stages of the scanner.

5.2 Linux kernel

Finally, the goal was, similar as in parts of Retbleed, to find suitable gadgets within the Linux kernel. To be able to compare the two scanners, we chose to run it on kernel version `v5.8.0-63-generic`, the same as used before. This specific kernel has a size of around 45 MB, which amounts to around 15 million potential gadget entry points. The constraints given to the scanner were identical. This included a fully controllable memory region from `%r14 + 0x8` up to `%r14 + 0x108` and some additional information about constant values of certain registers.

Given these constraints, *Inspectre Gadget* was able to locate two gadgets in the kernel:

(a) A double deref gadget which has already been found and used in the Retbleed paper:

```
<Gadget FR_DOUBLE_DEREF [0xffffffff813dc121:0xffffffff813dc13c]>
ffffffff813dc121  mov     rax, qword ptr [r14+0x28]
ffffffff813dc125  mov     rdx, qword ptr [r14+0x20]
ffffffff813dc129  mov     dword ptr [rdx+0x1c], 0xffffffff ; bogus
ffffffff813dc130  movzx   eax, byte ptr [rax+0x14]
ffffffff813dc134  add     rax, 0x1
ffffffff813dc138  mov     ecx, dword ptr [rdx+rax*0x4+0x58]
```

Listing 7: Double deref gadget in `mb_mark_used (fs/ext4/mballoc.c)`

(b) A newly, previously unknown store gadget:

```
1 <Gadget FR_STORE [0xffffffff813db126:0xffffffff813db153]>
2 ffffffff813db126  mov     rax, qword ptr [r14+0x28]
3 ffffffff813db12a  mov     rdi, qword ptr [r14+0x20]
4 ffffffff813db12e  movzx   eax, byte ptr [rax+0x14]
5 ffffffff813db132  lea     rdx, [rdi+0x58]          ; bogus
6 ffffffff813db136  mov     qword ptr [rdi+0x58], 0x0 ; bogus
7 ffffffff813db13e  add     rdi, 0x60
8 ffffffff813db142  lea     rax, [rax*0x4+0x8]
9 ffffffff813db14a  mov     qword ptr [rdi+rax-0x10], 0x0
```

Listing 8: Store gadget found by *Inspectre Gadget* in `mb_free_blocks (fs/ext4/mballoc.c)`

Gadget (b) is very interesting as its complexity demonstrates the power of the described techniques and tools perfectly. Not only is *Inspectre Gadget* able to see through the many arithmetic operations but also spot the dependency between `*secret` and `*rb` and how latter can compensate for operations applied to the dereferenced secret in order for the last `mov` instruction to be successful. We are able to exploit the gadget by initializing our memory regions as the following:

```
[r14+0x28] = *secret - 0x14
[r14+0x20] = *rb - 0x58
```

Listing 9: Initialization for store gadget found by *Inspectre Gadget*

Since the secret byte gets multiplied by `0x8` in the process, we will have to account for that dividing the predicted value by `0x8`. It is also important to take notice of the second bogus instruction on line 6 of Listing 8. This instruction will always cause a store at `rb[0]`, regardless of the secret byte. The extraction of the secret is still possible, we just need to account for this additional but predictable store.

5.3 Performance

The analysis was run on one of the education compute nodes from *COMSEC*. The node is made up of an "Intel(R) Xeon(R) Silver 4108 CPU @ 1.80 GHz" and has 16 GB of memory. Analyzing the mentioned Linux kernel took around a total of 18 h when doing a scan over all executable sections at a maximum gadget length of 20. This was done by using 16 processes in parallel, which, at least for this particular hardware, seemed to have been the most efficient settings, as can be seen in Figure 5.1.

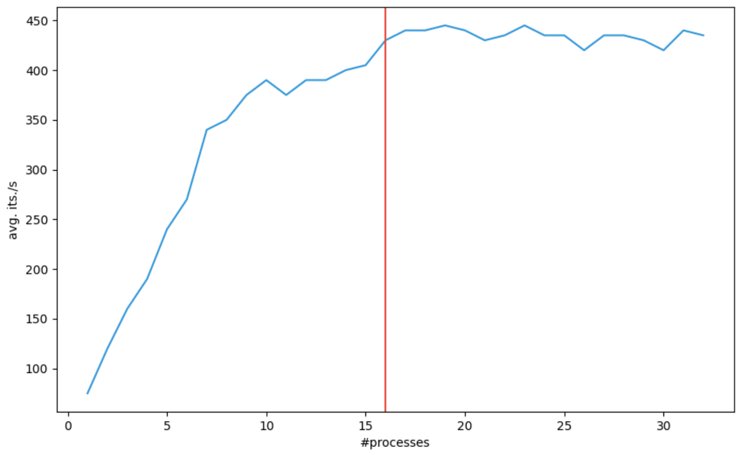


Figure 5.1: Iterations per second given number of processes

The results later revealed that a maximum gadget length of 10 would have sufficed to find the previously mentioned gadgets. This would have cut runtime nearly in half, at around 10 h of analysis in total.

Additionally, some profiling (see Figure 5.2) was done, which revealed that the majority of time is spent in *angr*'s functions. Only a very small fraction is used for AST simplification, analysis and gadget detection. Most of the time is spent in the symbolic execution engine and functions associated with it. Nearly half the time is spent on processing VEX blocks and their instructions, out of which, most is spent on memory related actions or *claripy* optimization. This indicates that either *Inspectre Gadget* is extremely well optimized or, more likely, a lot of processing time still goes to waste in – at least to us – mostly useless optimization in the background of *angr*. The following graphic is the call stack of the running program, making this difference even clearer. Width represents the overall percentage of time spent compared to other functions and the color indicates whether the function resides in *angr* or *Inspectre Gadget* (green: driver functions of *Inspectre Gadget*, red: *angr* library functions, yellow: *Inspectre Gadget* analysis).

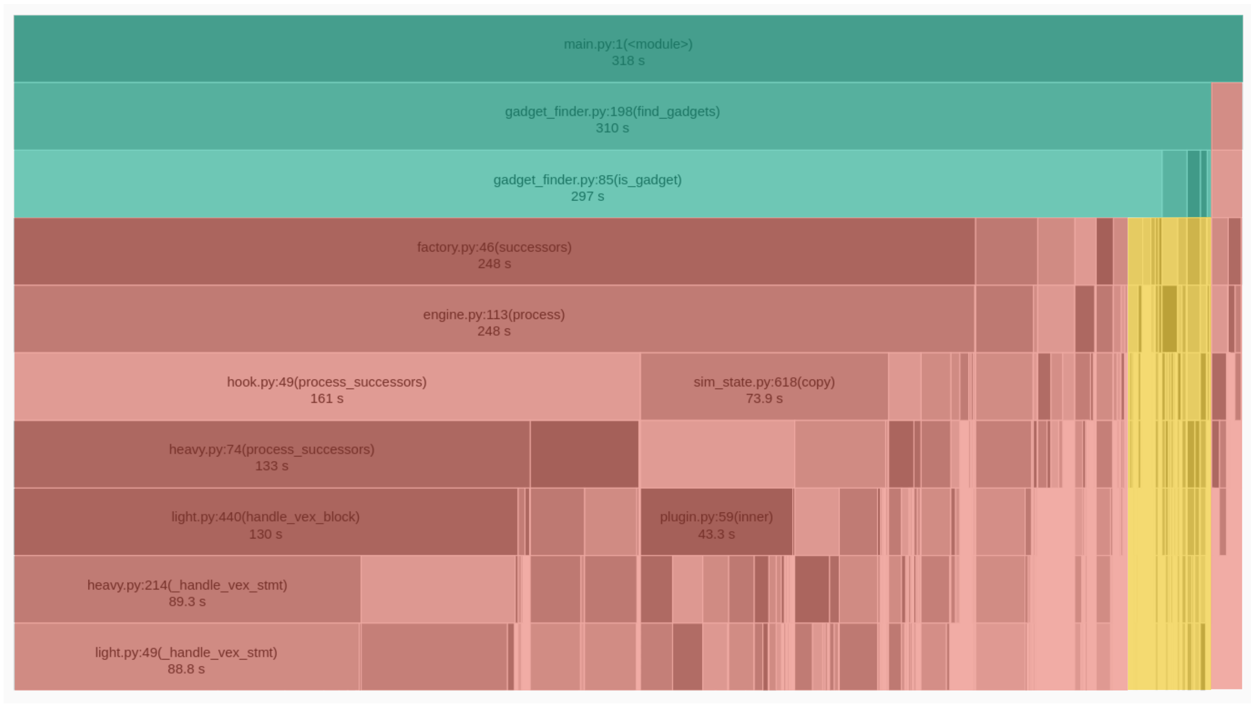


Figure 5.2: Profiling while running an analysis on the Linux kernel

There are still some obvious flaws with the system, which will be discussed in greater detail in chapter 6 Discussion.

Chapter 6

Discussion

6.1 Simplification and control analysis

As mentioned earlier, *Inspectre Gadget* is designed to make conservative guesses. This is especially true for the simplification and control analysis steps. The detection of gadgets could be improved by adding more edge cases of more operations and describing their effect on values in greater detail in terms of simplification and control analysis. There are many binary instructions that might allow for full control in certain cases (an `and` instruction here one operand is `MAX_INT`) instead of their current default behavior of assuming `PARTIAL` control. Obviously, going this in-depth would cause the project to open up to a near endless amount of potential improvements and could at some point come close to reimplementing an ISA from scratch. How much accuracy we would gain from implementing every last obscure instruction is debatable.

Another area from which we could gain more accuracy, if we were to improve it, is the merging of control analysis states – specifically the way required registers and memory regions are described. Currently, the common default behavior is to propagate the union of all the values coming from the different operators. The following example should illustrate the idea by showing the current behavior:

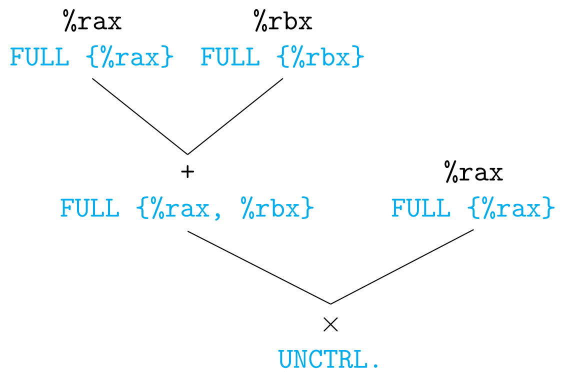


Figure 6.1: Example of current, slightly flawed, control analysis merging

It might seem strange for the analyzer to produce an `UNCONTROLLABLE` state for the final result. The reasoning behind this, however, is that by looking at the two operators of the multiplication, the analyzer will see that both require `%rax` to fully control the product. Since we can only guess at their relationship, we are forced to assume that one value will influence the other in ways that we cannot predict. What might be more accurate, however, is to distinguish between multiple ways of controlling a value, by storing multiple sets of registers per node (as opposed to only one), where each one of them describes a possible way to control the node. Looking at our previous example, we may find it possible to use either one of the two registers used in the addition to fully control the sum. During control analysis of the multiplication, we can then see that by using primarily `%rbx` to control the sum (and compensating for any offset caused by `%rax`) we will be able to control the complete value:

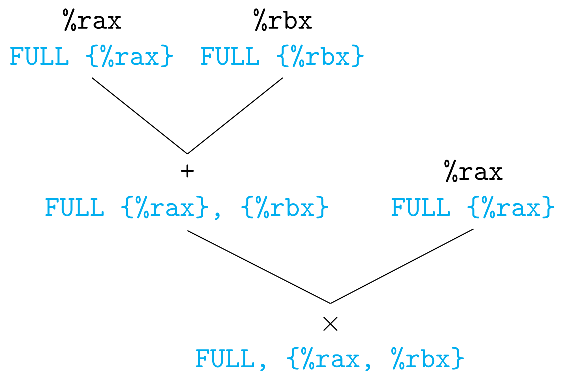


Figure 6.2: Example of improved control analysis merging

6.2 Symbolic Execution and *angr*

Finally, it is important to discuss the usage of symbolic execution. One major advantage of symbolic execution, over a maybe simpler method of analysis, is the usage of ASTs. They describe data flow in a very efficient way and allow for easy optimizations and simplifications. However, due to assembly being very well optimized in most cases these days, it is not always a given that ASTs built from code can provide much benefit through simplification. Symbolic execution also increases the memory footprint, as it stores way more information about a variable than just its concrete values. In our case, this increase in memory usage is completely negligible due to the fact that the analysis is always run on roughly 20 assembly instructions and therefore will never even have the chance to increase the size of an AST to such an extent that it would impact performance. But looking at the greater picture, it seems like symbolic execution simply provides enough benefits to outweigh the downsides.

Worth a more detailed discussion is, however, the framework or library used to complete the task. *angr* provides many benefits to binary analysis in the form of tools, but it must be noted that this might have even been one of the greater downsides in the context of this project. Due to messing with internal components, many symbolic executions will run into some sort of error, mostly due to forced concretization of our custom `Deref` object, which will obviously fail. It has been attempted to completely shut down parts of the framework in order to prevent unwanted behavior, but it has proven to be more difficult than expected. Additionally, while scanning the Linux kernel, it has happened more than once that execution suddenly came to a halt. Our best guess is that *angr* was neither designed to be run in parallel, nor have its functions called around 15 million times. The problem of halting could be circumvented by terminating all processes and restarting from an previous save point.

Not only did development time greatly suffer under the decision of using *angr* but also the performance of the scanner in general, which, as described in chapter 5 Evaluation, can cause the scanner to run for around 18 h on a Linux kernel. The choice of language might be worth discussing too, as Python might not be ideal for such an analysis due to its known issues regarding performance. As such, should this project be reattempted, it might be worth the consideration (mainly for the sake of performance) to use a different library (and maybe language all together) that isn't composed of as many interwoven parts as *angr* seems to have.

Chapter 7

Related Work

7.1 RETBLEED scanner

As mentioned before, the paper RETBLEED already employed the usage of a gadget scanner. This scanner was written in Python using the library *Capstone*. It works by propagating the `*secret` and `*rb` pointers through the different registers and checking whether we have any conflicting usage of the two when performing the final load/store. Using this technique, it was able to locate one gadget (see Listing 7) within Linux kernel `v5.8.0-63-generic`.

This form of register pollution tracking works great for most gadgets, but can break down on more complex ones. *Inspectre Gadget* circumvents this by using ASTs to not only track the movement of the pointers but rather also build the full tree of operations applied to them. The RETBLEED scanner is only designed to work on the x86 architecture, while *Inspectre Gadget*, thanks to its architecture independent design, is able to handle any architectures supported by *angr*. In addition, due to *angr*'s ability to handle control flow, should a gadget be "split in half" by a jump instruction, *Inspectre Gadget* will still be able to detect the gadget as such. It is only when a jump becomes ambiguous and no longer has a single potential target, where the scanner gives up and discards the trial, whereas the RETBLEED scanner will abort at any form of control flow instruction it encounters. This is illustrated by the fact that *Inspectre Gadget* was able to detect a second way of exploiting the RETBLEED gadget, by using a `jmp` instruction slightly farther down in the binary

```
ffffffff813dc1d0  jmp     0xffffffff813dc121
```

, which jumps to just above the gadget. This is quite useless as we can jump to the gadget straight away, but it shows that it would've still been able to detect a gadget, where one of the instructions for instance were to be located just before the `jmp` instruction instead, making it a two part gadget.

One point where *Inspectre Gadget* has a major drawback is speed. Symbolic execution can be very expensive and the way it is implemented in *angr*, it doesn't really improve the situation in any way. The RETBLEED scanner took around 6 min, which is a near 180-fold speedup compared to the 18 h *Inspectre Gadget* took.

7.2 Similar tools

There are a few tools which solve similar problems by for instance fuzzing [10, 5], dynamic taint analysis [12] or even symbolic execution[3, 4]. But despite the number of gadget scanners available, there is still a surprising lack of tools focusing on Spectre-BTI. Besides the two scanners presented up until now, there aren't any directly comparable, due to the difference in nature regarding the goal they are trying to achieve.

Chapter 8

Conclusion

We explained what speculative disclosure gadgets are, how they're structured and why they're important to our cause. Based on this knowledge, we showed how the concept of a gadget scanner using symbolic execution could look like. We discussed what libraries are suitable for such a task and showed how to utilize them together with some self-developed simplification and analysis algorithms. Finally, we ran our tool on a recent Linux kernel and discovered a previously unknown speculative disclosure gadget. Overall, we developed a tool which can locate speculative disclosure gadgets that can be used to leak arbitrary kernel memory from an unprivileged process.

Bibliography

[1] de Moura, L., and Bjørner, N. Z3: an efficient smt solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems* (March 2008), Springer, Berlin, Heidelberg, pp. 337–340.

[2] Fischer, S. Supervisor mode execution protection. In *NSA Trusted Computing Conference and Exposition* (2011).

[3] Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J., and Sánchez, A. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 1–19.

[4] Guo, S., Chen, Y., Li, P., Cheng, Y., Wang, H., Wu, M., and Zuo, Z. Specusym: Speculative symbolic execution for cache timing leak detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 1235–1247.

[5] Johannesmeyer, B., Koschel, J., Razavi, K., Bos, H., and Giuffrida, C. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In *NDSS Symposium 2022* (2022).

[6] Kadir, M. F. A., Wong, J. K., Ab Wahab, F., Bharun, A. F. A. A., Mohamed, M. A., and Zakaria, A. H. Retpoline technique for mitigating spectre attack. In *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)* (2019), IEEE, pp. 96–101.

[7] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2018).

[8] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 973–990.

[9] Nethercote, N., and Seward, J. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science 89*, 2 (2003), 44–66.

[10] Oleksenko, O., Trach, B., Silberstein, M., and Fetzer, C. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1481–1498.

[11] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference* (2006), Springer, pp. 1–20.

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

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

[14] various authors. Capstone - the ultimate dissasembler. `https://www.capstone-engine.org/`.

[15] Wikner, J., and Razavi, K. Retbleed: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security* (Aug. 2022). Intel Bounty Reward.

[16] Yarom, Y., and Falkner, K. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)* (2014), pp. 719–732.

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work:

Inspectre Gadget: Discovery of Speculative Disclosure Gadgets in the Linux Kernel

Thesis type and date:

Bachelor Thesis, December 7, 2022

Supervision:

Prof. Dr. Kaveh Razavi
Johannes Wikner

Student:

Name:	Loris Sikora
E-mail:	lsikora@student.ethz.ch
Stud.-Nr.:	19-921-956
Study Semester:	6

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the citation etiquette[1] information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

Place, Date

Loris Sikora

[1] https://ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf