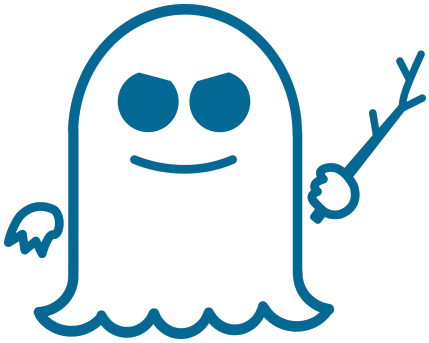
RUHR
UNIVERSITÄT
BOCHUM

RUB



RUHR-UNIVERSITÄT BOCHUM

Security Analysis of eBPF

Benedict M. Schlüter

Bachelor's Thesis – July 22, 2021.
Chair for System Security.

1st Supervisor: Prof. Dr. Thorsten Holz
2nd Supervisor: M.Sc. Sergej Schumilo

hgi SYSSEC

Preface

First of all, I want to thank Prof. Thorsten Holz and Sergej Schumilo for making this thesis possible and for giving me the opportunity to work on this topic. Also, I want to thank Piotr Krysiuk for working so closely on exploiting the Spectre vulnerabilities. Lastly, I want to thank Daniel Borkmann and Alexei Starovoitov for giving me insights into kernel development, the responsible disclosure process, and the strong involvement in the process of fixing the found vulnerabilities.

Abstract

eBPF is a revolutionary new OS technology; even Microsoft tries to implement it right now for Windows. However, the security of this technology has not been analyzed systematically. It is used by many large companies like Facebook, Google, and Netflix for various purposes since eBPF has a wide range of applications. It reaches from traffic inspection over tracing functionality to runtime security instrumentation. This thesis analyzes the security mechanism implemented by eBPF and bypasses certain restrictions. We use state-of-the-art fuzzing technologies and manual inspection. The thesis shows that eBPF is vulnerable to side-channel attacks and that unprivileged eBPF should be disabled by default on all security concerned Linux distributions. We also cover the countermeasures which can be applied to prevent speculative execution attacks.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 1
- 1.3 Contribution . . . 2
- 1.4 Organization of this Thesis . . . 2

2 Background **3**
- 2.1 Spectre Attacks . . . 3
  - 2.1.1 Speculative Bounds Check Bypass . . . 3
  - 2.1.2 Speculative Store Bypass . . . 4
  - 2.1.3 CPU-Architecture . . . 6
  - 2.1.4 Cache Coherence Protocols . . . 7
- 2.2 Architecture of eBPF . . . 8
  - 2.2.1 Overview . . . 8
  - 2.2.2 BPF System Call . . . 9
  - 2.2.3 Programs . . . 10
  - 2.2.4 Data Structures . . . 14
  - 2.2.5 BPF Verification Process . . . 15
  - 2.2.6 Spectre Mitigations . . . 17
  - 2.2.7 Program Attachment . . . 20
  - 2.2.8 BPF Type Format . . . 21
- 2.3 Fuzzing . . . 21

3 Methods **23**
- 3.1 Fuzzing . . . 23
  - 3.1.1 kAFL . . . 23
  - 3.1.2 Syzkaller . . . 24
- 3.2 Manual Inspection . . . 25

4 Implementation **27**
- 4.1 kAFL Agents . . . 27
- 4.2 Spectre Proof-of-Concept . . . 28
  - 4.2.1 Bounds Check Bypass Implementation . . . 29
  - 4.2.2 Store Bypass Implementation . . . 31

5 Fuzzing Results **33**
5.1 Path Coverage . . . 33
5.1.1 Program Loading . . . 33
5.1.2 Map Creation . . . 35
5.1.3 BTF Loading . . . 36
5.2 Findings . . . 36
5.3 Evaluation . . . 36

6 Spectre Vulnerabilities **39**
6.1 CVE-2021-29155 . . . 39
6.1.1 Attack . . . 39
6.1.2 Fix . . . 41
6.2 CVE-2021-34556 Reserved . . . 41
6.2.1 Attack . . . 41
6.2.2 Proposed Solution . . . 43
6.3 CVE-2021-33624 . . . 43
6.3.1 Attack . . . 43
6.3.2 Fix . . . 45
6.4 Speculative Store Bypass . . . 45
6.4.1 Attack . . . 45
6.4.2 Proposed Solutions . . . 46

7 Conclusion **47**
7.1 Summary . . . 47
7.2 Future Work . . . 47
7.2.1 Formal Methods . . . 48
7.2.2 New Fuzzing Techniques . . . 48
7.2.3 Advandced Microarchitectural Exploits . . . 49

A Acronyms **51**

B Appendix **53**
B.1 Working with eBPF . . . 53
B.2 Spectre Masking Example . . . 54
B.3 Spectre Proof of Concept Result . . . 54
B.4 BPF Structures . . . 56

List of Figures **61**

List of Tables **63**

List of Listings **65**

Bibliography **67**

1 Introduction

A new Linux kernel feature has become more and more fundamental for performance monitoring and network filtering in the past six years. We provide an overview of this newly developed technology and highlight why its safety is essential.

1.1 Motivation

Extended BPF (eBPF) is one of the latest major kernel features in Linux and is under rapid development. After replacing the old classic BPF (cBPF), eBPF is much more advanced and feature-rich. It allows user-supplied programs to be executed inside the kernel and carry out various tasks. The major applications are kernel tracing, policy enforcement, and network filtering. In all three categories, eBPF brings significant improvements because it makes things easier and safer to use. However, due to the extend of features, it is also a complex construct. It has its assembly-like language, which has similarities with a modern 64-bit RISC ISA. To allow unprivileged users to carry out network filtering tasks with eBPF, Linux 4.4 has introduced unprivileged BPF [1] so that a restricted eBPF program can be invoked without the need for `CAP_SYS_ADMIN`. Unprivileged eBPF programs make the security analysis attractive because exploits can lead to privilege escalation or information disclosure.

1.2 Related Work

Other researchers have already looked upon the eBPF security topic. They found interesting vulnerabilities that fall into three categories. The first class are classical programming issues, which can occur in every program. It includes all kinds of memory corruption issues and programming flaws like null pointer dereference and integer overflows. Fuzzers have found such bugs in the past. However, most reports came from individuals who manually inspect the code.

The discovery of the Spectre side-channel attack introduced a new class of exploits. BPF programs are executed in the kernel space, thus making Spectre-like attacks with unprivileged eBPF access possible since we do not have to cross any privilege boundaries. Piotr Krysiuk has found two vulnerabilities in late 2020 by exploiting

unprivileged eBPF access [2][3]. Initially, the first Spectre exploits against eBPF were developed by Jann Horn [4][5], who was one discoverer of Spectre.

The last class of bugs are logical flaws, where the verifier does not correctly check programs, and the range tracking fails. Here Manfred Paul [6][7] and Simon Scannell [8] have discovered issues within the verifier. This type of bug is unique in the operating system landscape; no other operating system has a static code analyzer build-in, which allows verifying and executing user-supplied programs.

1.3 Contribution

The thesis covers the current state of eBPF's security and its mitigations by explaining the security mechanism implemented by the developers. We also try to bypass security restrictions by using well-tested kernel fuzzers and manually analyzing code. The most successful attempt is trying to manually bypass specific security rules that the verifier has to ensure. Furthermore, we provide an in-depth analysis of vulnerabilities we discovered and give advice on the future. Finally, we helped to develop the fixes and reviewed the corresponding countermeasures.

1.4 Organization of this Thesis

The beginning of the thesis explains how two Spectre attacks work and give necessary information about the central processing unit (CPU) design to understand the exploits in the later sections. The first section also gives an overview of eBPF, how it works, and its use cases. Mainly, we explain the eBPF verifier and its security mechanism. The following section covers the methods used to analyze the security and why they got chosen. The fourth section is about implementing the fuzzing agents and how they are designed considering the specialty of the eBPF syscall. The same section also covers the Spectre proof-of-concept code, which is the basis for the proof-of-concept codes of the Common Vulnerabilities and Exposures (CVE) discovered. The subsequent section is about the fuzzing results and evaluates them. The sixth section covers the results of the manual inspection and explains two CVE's in more detail. The same section also introduces two additional bugs, which got a CVE reserved. Finally, the last section concludes this thesis by summarizing the main points and giving ideas for future work on eBPF security.

In this thesis, we use the terms eBPF and BPF interchangeably.

2 Background

In this section, we cover the necessary background to understand the vulnerabilities which were found. The goal is not to explain the whole eBPF ecosystem; instead, only the relevant information is provided, which is essential for understanding the root cause of the vulnerabilities. Finally, we introduce the CPU model, which we use throughout the thesis. Further, we cite significant changes to eBPF and the introduced features with the corresponding Linux commit.

2.1 Spectre Attacks

It is vital to understand the theoretical concept of the Spectre attacks as a prerequisite for the proof-of-concept code in the later chapters. Therefore, we briefly introduce Spectre v1 and v4 in this section. Further, for an advanced understanding, it is necessary to be familiar with the inner working of a CPU. Thus, we also introduce a cache coherency protocol and the high-level CPU architecture we assume in this thesis.

2.1.1 Speculative Bounds Check Bypass

Speculative bounds check bypass, also known as Spectre v1, was one of the initial speculative execution vulnerabilities discovered [9]. The root cause behind the attack is that to speed up the execution, modern CPUs execute instructions out-of-order and speculate about the outcome of instructions by relying on the history of the last executions. If the speculation is correct, it will result in a massive speedup; if the prediction is wrong, the pipeline will be flushed, and the CPU continues execution at the point before the speculation happened. With the flush, the CPU also restores all registers to the state before the speculation.

However, some structures within a CPU are not reset to the pre-speculation state, for example, the cache. To be more precise, the values which get newly loaded into the cache remain in it. Of course, if the CPU wrongly updates a memory location speculatively, it will be reverted. Otherwise, the wrong value eventually stays in the cache and will be later miswritten to memory. However, this is not the case for load operations. If a load operation is carried out speculatively, the address is loaded into the L1 cache. If the resulting load is wrongly carried out, the value stays in the

cache since we did not manipulate any data. The only difference is that the value is now faster accessible from the CPU core. This behavior is the side channel that is used by Spectre gadgets to exfiltrate data. Listing 2.1 shows an example how this looks for C code. For this exploit to work, we have to make assumptions that can be achieved within real-world code. However, to keep it simple, we have trimmed the code down to the essence.

```
1 uint8_t leak_array[256];
2 uint8_t out_of_bounds_array[256];
3
4 if (index < *max_index)
5 {
6   leak_array[out_of_bounds_array[index]];
7 }
8 [...]
```

Listing 2.1: Spectre v1 example

These assumptions are that the attacker has access to the variable `leak_array` and can measure the time or the clock cycles it takes to access elements within it. When the comparison for the if-statement in line 4 is executed, the integer, which `max_index` points to, is not in the cache. Otherwise, the CPU speculation would not reach far enough if the condition in the if-statement is not fulfilled. All elements within `leak_array` are not in the cache at the beginning of the execution. This is necessary since it is required to measure the access time and behold which value of the array was loaded into the cache.

Furthermore, for successful exfiltration of the data, the branch predictor unit must first be trained to execute the code within the if-statement. We achieve this by running through the if-statements multiple times with the `index` value in-bounds. If the index value is out-of-bounds and the branch predictor was trained to take the branch, the instructions in the if-statement will be speculatively executed since the integer to which `max_index` points to is not loaded into the cache and thus not fast accessible. The cache is now manipulated that it is possible to measure the access times and obtain the value, which was stored on `out_of_bounds_array[index]`. Where `index` should point to a value that is larger or equal than 256; otherwise, we could also read the value in the non-speculative domain. How such an exploit looks for a practical example is explained in Section 4.2 and Chapter 6.

2.1.2 Speculative Store Bypass

Speculative Store Bypass, also known as Spectre v4, was discovered as follow-up research after the initial Spectre vulnerabilities were disclosed. It was discovered by Microsoft [10], and Google Project Zero [5]. To execute as many instructions

as possible, the CPU tries to prevent as many data dependencies. For example, the write-after-read (WAR) and write-after-write (WAW) dependencies could be resolved by renaming registers. Especially Tomasulo's algorithm [11] lays the theoretical basis for such optimization techniques.

The only dependency which slows down the CPU immensely is the read-after-write (RAW) dependency. Before the CPU has to carry out the read instruction, the write and all instructions which the write depends on are executed. In some cases, e.g., the offset to a pointer must be loaded from memory, the CPU cannot detect these dependencies. To not stall, the CPU speculates that no RAW dependency exists between two or more instructions and executes the load instructions and everything without a dependency to the store speculatively, without knowing for certain if they are independent. If the prediction is correct, it will result in a massive speedup; if the prediction is wrong, the penalty will be relatively low compared to the performance gain. The CPU then has to recover the register states and re-execute the dependent instructions. However, this behavior can be used to manipulate the cache, which allows the execution of a Spectre gadget and obtain the side-channel information. Listing 2.2 shows a simple example.

```
1 pointer = secret;  //set pointer to malicious value
2 [...]
3 pointer = array_ptr; //write gets skiped due to slow address
4 value = *pointer;  //speculative derefenrece malicious pointer
5 cache_trace[value]; //manipulate the cache
```

Listing 2.2: Spectre v4 example

In line 1, the pointer is set to a malicious address, which we cannot dereference. The store in line 3 is speculatively skipped because the variable `pointer` is manipulated to depend on previously slowly-loaded data or the store queue is under high pressure, and no RAW dependency can be detected. Slowly-loaded data mean data that do not reside in the cache. The load in line 4 is executed speculatively since the CPU speculates no dependency between the store in line 3. The speculatively loaded value is then used in line 5 to manipulate the cache. Another part of the exploit code measures the access time to the cache regions, and we obtain the speculatively loaded secret value. As in the previous example, we assume that all elements from the array `cache_trace` do not reside in the cache when the code is executed.

It is crucial to notice that the cache gets manipulated a second time if the CPU re-executes the instructions when the RAW dependency gets resolved. If we try to obtain the information, we will have two array elements residing in the cache. To decide which element is the secret, we first have to determine the actual value used. The code snippet in Listing 2.2 does not work out of the box. It is necessary to adjust the addresses and the array index for the exploit to work. However, for simplicity,

we only cover the base version. In Section 6.2 we introduce a working Spectre v4 exploit.

2.1.3 CPU-Architecture

In our CPU model, we assume we have a CPU with three different cache levels, where each cache contains a subset of the higher-level cache. The smallest cache is the L1 cache, divided into an L1 instruction and an L1 data cache. The intermediate cache is the L2, which is also core exclusive. The last-level cache, also known as LLC or L3 cache, is shared between all cores on the same core/compute complex (CCX). More advanced CPUs may have different CCXs combined using a interconnect technique as AMD is doing it [12]. However, all exploits in this thesis are tested using Intel processors from the `Haswell` and `Kaby Lake Refresh` generations. The topology, as seen in Figure 2.1 is assumed when we are talking about the exploits and CPUs. It is the topology for nearly all desktop CPUs for the last ten years.

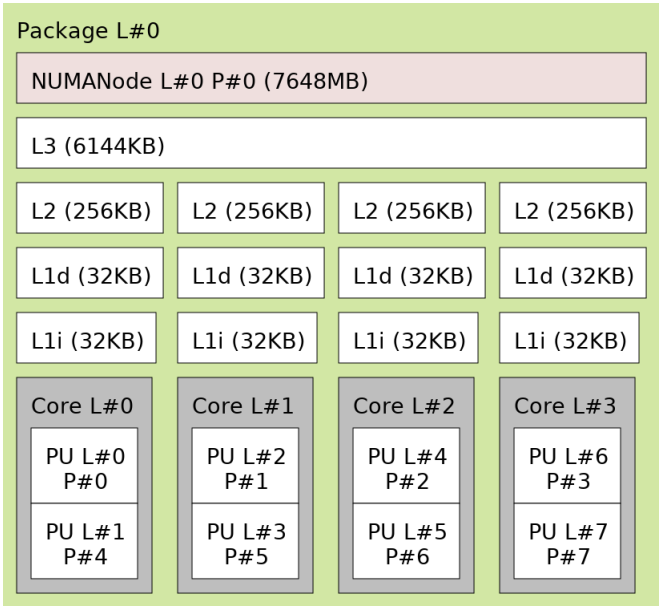


Figure 2.1: Intel Core i5-8250u topology

Further, another important aspect is that modern processors are, in theory, heavily bottlenecked by memory access because the access latency for memory operation has not been increased significantly for years [13]. To increase the performance, modern processors rely on cache technologies and execute instructions out-of-order. Some out-of-order instructions are speculatively executed, meaning the CPU does not know if these instructions are executed in the non-speculative domain. If the speculation is correct, it results in a massive speedup and a relatively low penalty if

it is wrong. In that case, the CPU state is rollbacked, and all registers are reset to the pre-speculative values.

Modern processors speculate about the outcome of conditional jumps if the data for the comparison is not in the cache and execute the following instructions speculatively. Intel Skylake CPUs can execute up to 224 operations out-of-order, translating to roughly 350 μ-operations.

2.1.4 Cache Coherence Protocols

It is necessary to have a protocol that ensures coherence between the different caches of the system. Intel has used the MESIF protocol in the past [14], while AMD has used the MOESI protocol [15]. The difference between those protocols is that the MOESI defines an additional `O` state for a cache line and the MESIF and additional `F` state. The newer manuals do not refer to a specific protocol; instead, they explain the behavior of the coherence [16]. Both major AMD64 CPU manufacturer coherency protocols rely on the MESI protocol [17] and extend it with additional states. The MESI protocol differentiates between 4 different cache states.

M. The cache line is `modified`, meaning the value in the cache differs from the copy within memory. If the line is repressed out of the cache, it has to be written back to memory. If another cache reads the memory, it is written back, and a transition into the shared state happens.

E. The cache line is `exclusive`, meaning the line is only accessed by the current cache and is not modified.

S. The cache line is `shared`, two or more cores share the same line, which contains the same value within all caches.

I. The cache line is `invalid`, no data can be obtained from it, the line is write-only.

Table 2.1 shows which cache line can have the same state on two cores.

Table 2.1: MESI concurent states table

	M	E	S	I
M	-	-	-	+
E	-	-	-	+
S	-	-	+	+
I	+	+	+	+

As the Intel documentation was not as detailed as the one from AMD, only the cache coherency of the newest `Zen 3` architecture from AMD is covered.

This architecture has an inclusive L2 cache and a write-back L3 cache. Where each core has its L1 and L2 cache, multiple cores have a shared L3 cache. It means that each cache entry within the L1 cache also resides in the L2 cache. If a value gets replaced in the L2 cache, it will be written back into the L3.

The L3 cache maintains so-called *shadow tags* for each L2 cache value in the complex, meaning if another core misses the L2 and L3 cache, the L3 cache consults the *shadow tags*, and if successful, initiates a cache-to-cache transfer. So the L3 cache is used for synchronization between the different caches of the CPU.

If the CPU misses the L1 and L2 cache, and the data resides within the L3, the average load-to-use transfer rate is around 46 cycles [16]. We can assume that Intel uses a similar technique and the latency is roughly the same.

This latency is essential since it determines the speculation window we obtain for Spectre exploits against eBPF.

2.2 Architecture of eBPF

Before introducing the vulnerabilities, we give an overview of the eBPF technology. It is necessary to understand how exactly those vulnerabilities arise. Thus this section shows how eBPF works, what types of programs we have, and how the system call executes. Important security aspects are also covered. BPF stands for Berkley Packet Filter. However, the current implementation has little in common with the old implementation classic BPF; thus, the name BPF should not be traded as an acronym.

2.2.1 Overview

eBPF is a unique technology in operating systems. It allows user-supplied programs to run inside the kernel space and provides a lightweight alternative for certain kernel modules. Also, it is possible to instrument the whole kernel with eBPF and hook into many functions with kprobes or tracepoints while also allowing for fast network filtering. It is achieved by filtering packets before they arrive at the kernel network stack. The technology for this is called Xpress Data Protocol (XDP). While programs alone were not sufficient to carry out all tasks, many extensions to the system call have been made. For example, to store data that shall be persistent over several runs and communicate with the userspace, BPF maps have been introduced. Navigation through structs, which are not defined in the kernel headers, is possible through BPF Type Information (BTF). Not all options are covered in-depth, only the ones that are essential for understanding the technology and the attacks that are later covered.

2.2.2 BPF System Call

The BPF system call does not only allows us to execute and load programs. It is also used for far more cases. A full list can be found in Table B.6. These options are the first parameter for the system call. The second contains the actual data used in the functions that the large switch-case statement executes within the system call handler in the kernel. It can be found under `linux/kernel/bpf/syscall.c`. The data structure is a union in which multiple structs for the different system call options are defined. Only one struct at a time can be used. Consequently, a union is the ideal data structure here. The C wrapper for the system call is shown in Listing 2.3.

```
1 #include <linux/bpf.h>
2
3 int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Listing 2.3: BPF system call wrapper

The system call number varies between architectures; it is defined under the macro `__NR_BPF`, and for the x86_64 architecture, the corresponding integer is 321 and for ARM64 280. The BPF system call can be disabled in the kernel config. With Linux 5.13, all BPF configuration options became grouped. The configuration folder can be found under `General Setup` for enabling the BPF system call and tune other BPF related configurations. Depending on the configuration, several options can be selected. The BPF code can be just-in-time (JIT) compiled or interpreted which is defined under `net.core.bpf_jit_enable`. This configuration option can have three values.

0. BPF JIT is disabled; the BPF code is interpreted, which is pretty slow compared to the other option.
1. BPF JIT is enabled; after successfully passing the verifier, the program gets JIT compiled and loaded into the kernel.
2. BPF JIT is enabled; after successfully passing the verifier, the program gets JIT compiled and loaded, but the compiled code is also written into the kernel ring buffer.

If the kernel option `CONFIG_BPF_JIT_ALWAYS_ON` is set to true, we cannot change the value above within the running kernel. Other options influence the system call as well. `CONFIG_BPF` enables BPF inside the kernel without the system call. The reason for that is that BPF was first used internally by the kernel without a system call interface. Later the option `CONFIG_BPF_SYSCALL` was added, which allowed loading eBPF programs through a system call. The option `BPF_JIT` has to be enabled to compile the kernel with the JIT compiler. It relies on either `CONFIG_HAVE_EBPF_JIT` or `CONFIG_HAVE_CBPF_JIT` being true. The latter one is depreciated. Finally, to enable

some particular program types, e.g., kprobe or perf events, the option `BPF_EVENTS` has to be set to true.

Another interesting feature that BPF is capable of is runtime instrumentation of the Linux Security Module (LSM). It could potentially replace SElinux and Appamor since it is more flexible than these two tools and has a wider feature set. To enable eBPF LSM hooks, the kernel option `BPF_LSM` has to be true, which depends on `BPF_EVENTS` and `BPF_JIT` being true.

2.2.3 Programs

The BPF system call can be invoked with many command arguments; the main parameter is `BPF_PROG_LOAD`. It creates a BPF program that loads into the kernel. This section shows how to construct these programs and which limitations they have. Notice that a program does not run until it is attached to an event, and this event gets triggered.

Syntax & Semantics

eBPF has its assembly-like language, which has similarities with a restricted 64-bit RISC instruction set. The language consists of opcodes for arithmetic, bit-operation, memory access, and branch instructions [18]. Figure 2.2 shows the instruction encoding of the 64-bit instruction format. The opcode-field can be further divided into the

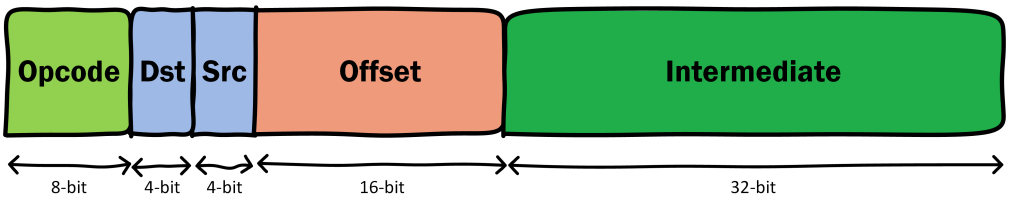


Figure 2.2: Instruction encoding eBPF assembly

different opcode classes and source types, i.e., intermediate or register. The endianess of the instruction encoding is determined by the endianess of the host architecture. For x86_64, it is little-endian. Even if the instruction set architecture (ISA) is 64-bit, we can also run eBPF on 32-bit machines due to changes to the JIT compiler [19]. However, 32-bit transition happens in the JIT compiler, and everything else is identical to 64-bit systems. The instruction format can be translated into a struct to directly generate bytecode in C macros without a specific compiler for the BPF language. Despite making things, easier LLVM has released updates to clang that allow us to compile C code into BPF bytecode.

The assembly language has eleven registers in total, where one register is the read-only frame pointer, so we have only ten left for general purpose usage. Figure 2.3 shows the calling convention. Registers 1-5 may contain sensitive kernel data after

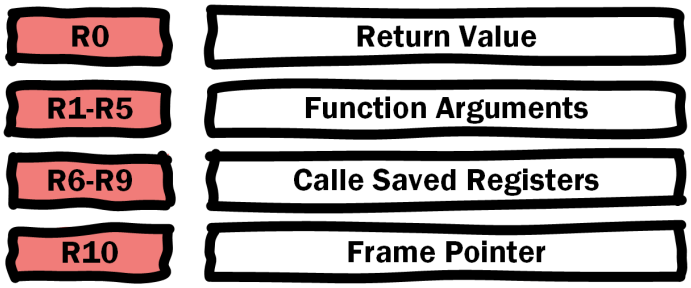


Figure 2.3: Calling convention eBPF assembly

a function call because they could be used inside the function. While registers 6-9 mainly serve the purpose to save variables before a call is made. A 512-byte stack provides non-persistent storage for any eBPF program if it is necessary to store more data. This size is fixed and can only be changed within the source code of the kernel. Maps are used to get access to more storage. We will cover maps in-depth in Section 2.2.4. It is essential to notice that maps are accessible from the userspace through system calls, while the stack is only read and writable from within a BPF program. It is also important to note that the stack is not consistent over several runs of the BPF program, so it cannot be used to count events over multiple program executions.

Capabilities

To access the entire feature set of BPF programs `CAP_SYS_ADMIN` is needed (actually only a subset since this capability is overloaded). For a more restricted usage `CAP_BPF` was introduced, which provides more access than invoking the system call without any capabilities but without giving the user superuser privileges as `CAP_SYS_ADMIN` is doing it. Nevertheless, it is also possible to load BPF programs without these capabilities if the kernel parameter `kernel.unprivileged_bpf_disabled` is set to zero. So we distinguish between unprivileged BPF and privileged BPF, although the process of loading a program lies in between these extreme cases since capability information is used for the safety checks.

`CAP_BPF` provides functionality like `CAP_SYS_ADMIN` on the BPF side. However, to attach programs to a critical event or run the programs, the related capability for the attachment is needed. For example to attach an XDP program to a socket additional `CAP_NET_ADMIN` is needed [20]. For later, it is vital that only bugs in unprivileged programs can be used for exploitation. Still, it is also interesting to find bugs in the privileged case for stability reasons. The main difference between

those modes is that a privileged program has access to all helper functions and can be attached to all events which BPF offers, e.g., kprobes, tracepoints, or perf events. While unprivileged programs can currently have only the types `socket_filter` or `cgroup_skb` and thus cannot be attached to kprobes. Another difference is that unprivileged programs have to pass more verifier checks. The first one we have already mentioned is that it is not possible to call all helper functions, but also the restrictions seen in Listing 2.4 are applied.

```
1 env->allow_ptr_leaks = bpf_allow_ptr_leaks();
2 env->allow_uninit_stack = bpf_allow_uninit_stack();
3 env->allow_ptr_to_map_access = bpf_allow_ptr_to_map_access();
4 env->bypass_spec_v1 = bpf_bypass_spec_v1();
5 env->bypass_spec_v4 = bpf_bypass_spec_v4();
6 env->bpf_capable = bpf_capable();
```

Listing 2.4: Verification restrictions from `linux/kernel/bpf/verifier.c`

As an unprivileged user, it is not allowed to store pointers on maps. Otherwise, they could be leaked to userspace and be used to bypass kernel address space layout randomization (KASLR). It is also prohibited to read values from the uninitialized stack, but this restriction also applies to privileged programs. Spectre mitigations are applied if we do not own the capability of reading arbitrary kernel memory. If we have the capability, we can read kernel memory without BPF, so there is no reason to prevent Spectre attacks. The call for `bpf_capable()` checks whether we have higher privileges, i.e. `CAP_BPF` or `CAP_SYS_ADMIN`. The aforementioned is necessary since, in some cases, one of these capabilities is needed to execute the system call successfully. For example, to load BTF data into the kernel with the corresponding system call argument. Contrarily, the kernel will reject the system call and emit the error message `EPERM` if we do not own the permissions.

To summarize, `CAP_BPF` gives us superuser permissions on the eBPF side, which translates to that we can execute any eBPF system call. To attach the programs to restricted events, we need the permissions for the respective subsystem, i.e., `CAP_SYS_PERFMON` for tracing related attachments and `CAP_NET_ADMIN` for network-related ones. With `CAP_SYS_ADMIN` no restrictions apply.

Loading Process

Related to Section 2.2.2 this section visualized the process of loading a program in the kernel. Figure 2.4 shows the loading process of a BPF program and also simple interactions with the userspace.

After the system call is invoked, the kernel copies the program from userspace into kernel space and checks if it is safe. After the verification process has finished successfully, the program gets either JIT compiled or interpreted depending on the kernel

configuration. All major Linux distributions JIT compile the code since it is much faster than interpreting it. After the program has been compiled, it lies in the kernel space. Another system call is executed, attaching the program to a corresponding event. For example, all programs with the type `socket_filter` can be attached to any socket on the system. In contrast, a program with type kprobe can only be attached to kprobes, to be more precise, to a perf event referring to a kprobe. If the corresponding event, e.g., the hooked function gets executed, or a packet arrives at the socket, the eBPF program is executed non-preemptive, meaning no interrupts can intercept it while running. Notice that there is some work allowing certain program types to run with preemption, so it might change in the future [21]. After the program has run, it exits, and the execution of the kernel code continues. The programs can manipulate the data they receive with their return value. The data is highly dependent on the program type.

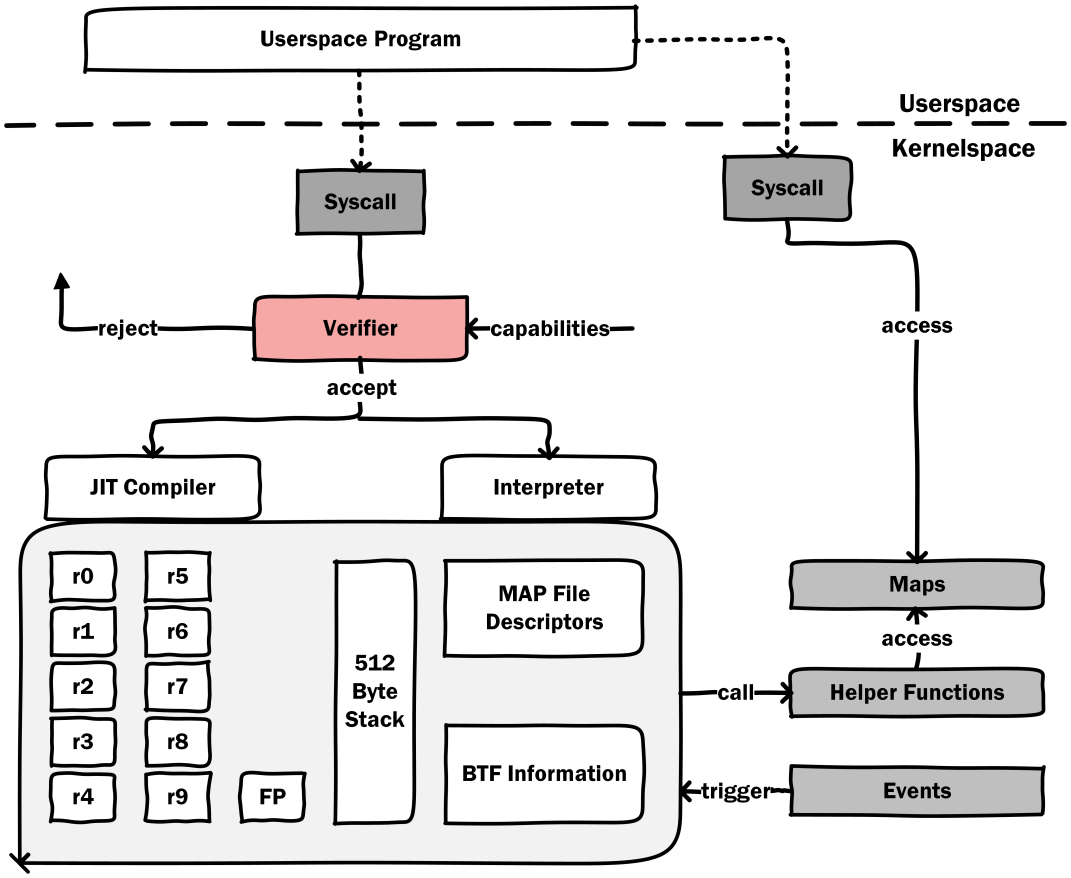


Figure 2.4: Loading process of an eBPF program

The verifier also rewrites code to apply Spectre mitigations, depending on the capabilities of the calling process. The struct `bpf_prog_aux` in `include/kernel/bpf.h` holds information about the eBPF program itself, maps that are associated

with the program, and BTF information, used in the program. The struct also contains other information related to the program, which is not relevant in the context of this thesis but relevant for further understanding of the eBPF subsystem.

2.2.4 Data Structures

The stack is only 512 bytes in size; multiple format strings can easily exceed that limit and cannot be used. However, to solve the problem and allow communication with the userspace, BPF has introduced many data structures which reside under the global name **map**. A map can be an array of integers, a linked list, or an array of perf events. Table B.4 contains an entire list of possible types. The `BPF_MAP_TYPE_ARRAY` map is covered more in-depth. As with the program helper functions, not all maps are usable for unprivileged programs, and some maps are only accessible with some program types.

Multiple BPF programs can use one map. Information about which map is connected to which program is stored inside the struct `bpf_prog_info`. The abovementioned struct is linked to the corresponding program.

The `BPF_MAP_TYPE_ARRAY` is the generic structure for storing data. As explained in Section 2.2.2, the system call expects as the second argument a `bpf_attr` union, in which the corresponding struct is filled. Listing 2.5 shows which parameters in the struct are used for map creation. The map type is here `BPF_MAP_TYPE_ARRAY`. The `key_size` specifies how long the key is in bytes. The key size limits the `max_entries` in a map. If we take a one-byte key, we can only access 2^8 elements. As a rule of thumb, a key size of 4 is taken for most programs. Theoretically, we can access far more elements with a larger key size, but this is not suitable for most cases and an unnecessary waste of resources.

```
union bpf_attr {
  struct { /* anonymous struct used by BPF_MAP_CREATE command */
    __u32  map_type;      /* one of enum bpf_map_type */
    __u32  key_size;      /* size of key in bytes */
    __u32  value_size;    /* size of value in bytes */
    __u32  max_entries;   /* max number of entries in a map */
    [...]
  };
[...]
};
```

Listing 2.5: `BPF_MAP_CREATE` struct

Inside a BPF program, we cannot access the maps directly. If we want to lookup an element, we have to push the corresponding index on the stack and call the

`bpf_map_lookup` function. The address of the element location is then stored in r0. The pointer r0 is used to navigate to a specific location in the entry. Figure 2.5 shows how this works; the verifier ensures that the pointer to one array element is in-bounds, and we can only add `value_size-1` to the pointer. In this instance, we can only add an offset of 39 to the element pointer.

To put it simple, we can think of the `BPF_MAP_TYPE_ARRAY` as a sequential array of arrays, where the `value_size` determines the inner array size and `max_entries` the outer size. Other maps have completely different layouts, for example the ring buffer.

Furthermore, maps can be accessed through the eBPF call to the previously mentioned function from within the BPF program or using the BPF system call from a userspace program.

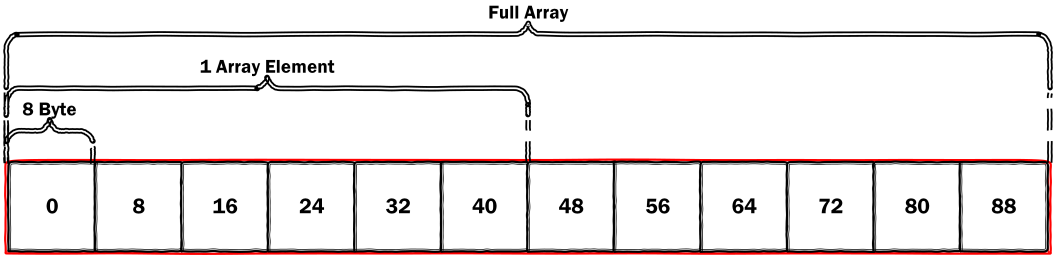


Figure 2.5: `BPF_MAP_TYPE_ARRAY` layout

2.2.5 BPF Verification Process

To ensure the program does not crash and does not access memory that it is not allowed to, the verifier checks the program before it is loaded into memory. If it fully fulfills certain safety conditions, the verifier will accept it. On the other hand, if the previous conditions are not met, the verifier will reject it, and the system call fails to execute. As previously seen, the capabilities define which safety conditions the program has to fulfill. The two extremes are the unprivileged user, who has every restriction enabled, and the programs that run with `CAP_SYS_ADMIN`, which only get checked for a basic set of properties.

Internals

The BPF verifier is a static code analyzer. At the beginning of the verification procedure, it performs a depth-first-search and rejects all programs that are not directed acyclic graphs. It follows that no instruction should be reachable from itself, so there are no loops. The loop restriction got changed with commit [22], where this restriction only applies to unprivileged users. The function `check_cfg`

within `kernel/bpf/verifier.c` is responsible for this task. It is also forbidden to have unreachable instructions in the code. It is only half true. There are programs where a jump instruction is always taken; thus, the code beneath the jump will never get executed. If the verifier detects this case, the program will not be rejected. Instead, the dead code is replaced by *jump program counter -1* instructions. There is a limit that only `BPF_MAXINSNS` (currently 4096) can be processed for unprivileged users and `BPF_COMPLEXITY_LIMIT_INSNS` (currently 1 million) for privileged ones. This does not translate 1:1 to program size since this limit defines the instructions which the verifier can process. If the verifier can prune large parts of the program, it can process much more instructions. On the other hand, if the program contains many branches, the limit might be more tighten [23].

In the main verification procedure, we loop over every execution path possible and check the register states for each possible state. Here the verifier distinguishes between different register states, which change depending on the state of the program. The entire list can be found in B.5. `SCALAR_VALUE` is the most used one. It is just an integer that has no limitations, it can be stored in maps, and every arithmetical or logical operation is possible with these values. The verifier has a complicated range tracking system for those values. Every time a conditional jump is taken based on this register, the possible values of the register are updated. It is only possible to add a `SCALAR_VALUE` to a pointer if the register range is in bounds. If arithmetic operations are performed on those registers, the verifier will keep track of the new values. If this range tracking system gets broken, an unprivileged user can access arbitrary kernel data.

However, not for all operations range tracking is performed, for example, if we negate a variable, all information is lost, and the variable is marked as an unknown integer. The currently supported tristate number (tnum) operations can be found in `kernel/bpf/tnum.c`. These are at the time of writing addition, subtraction, multiplication, and all boolean operations. The range tracking can also be performed on min and max values. However, these operations can only be obtained from the verifier source code. The min-max values can be derived from the tnum struct, which holds information about bits set to one and bits which value we do not know. Combining these pieces of information, we can derive any signed or unsigned min-max 64-bit and min-max 32-bit values, which a register can hold. The other way around is also possible. We can obtain the tnum value from min and max values for the specific register width.

The second important class is `PTR_TO_MAP_VALUE`, which points to an element in an array. If the element is larger than one byte, we can add offsets to the pointer and use it like a regular pointer in C. The offset can be static, so the verifier knows which value gets added to the pointer or dynamic, where the offset lies between the min and max values obtained from the verifier. In the last case, the verifier ensures that the access will be in bounds and rejects all programs which violate the restrictions. It is

also forbidden to have mixed signed and unsigned bounds dynamic offsets within a pointer.

The verifier itself is a dynamic construct many lines change over time as seen in Figure 2.6. The information was obtained using the git diff functionality between different tags. The command to get the last table entry is `git -no-pager diff -numstat v5.11..v5.12 kernel/bpf/verifier.c`.

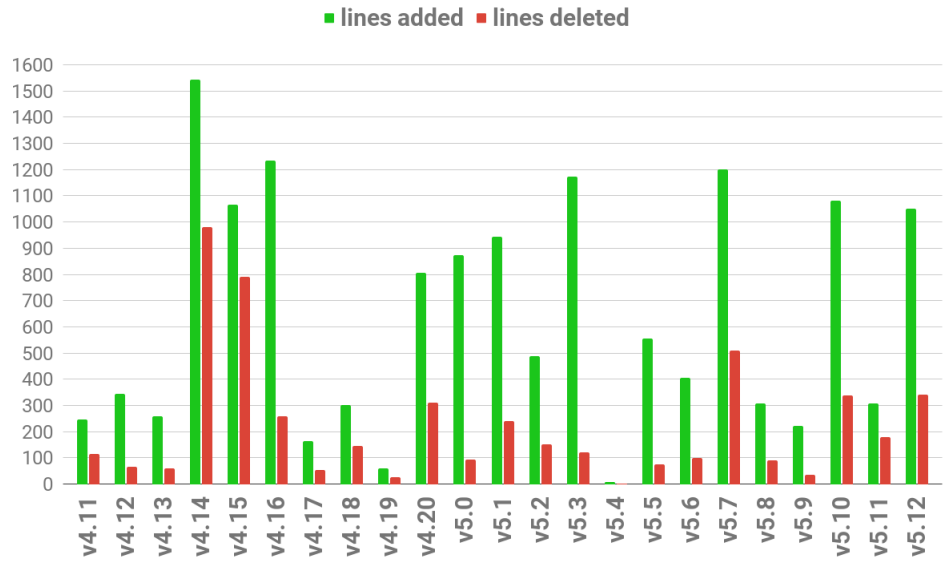


Figure 2.6: Verifier lines changed

2.2.6 Spectre Mitigations

BPF allows unprivileged users to run code inside the Linux kernel. While Spectre variant 3, also known as Meltdown, was used to cross user-kernel space boundaries, this is not necessary in this case. Our program runs inside the kernel, so we do not need to cross any privilege boundaries. Thus, the most critical variants are bounds check bypass, also known as Spectre v1, and speculative store bypass, also known as Spectre v4. The verifier has implemented mechanisms to prevent these attacks.

Bounds Check Bypass

Against bounds check bypass, a pointer sanitization mechanism is implemented after Jann Horn pointed out the issue with eBPF [24]. Consider the code found in Listing 2.6 as example.

```
1 (79) r2 = *(u64 *)(r7 +8)
2 (79) r3 = *(u64 *)(r0 +4608) //slow read
3 (57) r3 &= 1
4 (47) r3 &= 2
5 (2d) if r2 > r3 goto pc+7 //bypass safety check
6 (0f) r4 += r2
7 (71) r4 = *(u8 *)(r4 +0)  //access value OOB
```

Listing 2.6: eBPF Spectre v1 example

In the first line, the malicious offset value is loaded into r2. It is done using a fast pointer that resides inside the L1 cache. In the second line, we load a value through a slow pointer; this means the data, where the pointer points to, is not in the L1 nor the L2 cache of the CPU. How this can be achieved is covered in Chapter 4. On the loaded value, two logical `ands` are performed with 1 and 2, so r3 is 0 in line 4. After that, an unsigned comparison happens; if r2 is greater than zero, we skip the following seven instructions.

We assume that the value r3 is not loaded in time, and the CPU executes instructions 6 and 7 speculatively. Following this, r2 could be much greater than 0, and when added to the pointer r4, it could be an out-of-bounds memory access. The speculative loaded value in r4 is used to manipulate the cache. Later we measure the access time to that cache region and obtain the speculatively loaded data.

A pointer masking scheme has been developed against Spectre v1 attacks in general [25]. It prevents arbitrary memory loads and has been applied to eBPF programs. The pointer masking can be seen in Listing 2.7. First of all, we check if the offset is negative or positive and change the sign of the offset register to a positive value.

```
1 if(off_reg < 0)
2     BPF_ALU64_IMM(BPF_MUL, off_reg, -1);
3 BPF_MOV32_IMM(BPF_REG_AX, aux->alu_limit);
4 BPF_ALU64_REG(BPF_SUB, BPF_REG_AX, off_reg);
5 BPF_ALU64_REG(BPF_OR, BPF_REG_AX, off_reg);
6 BPF_ALU64_IMM(BPF_NEG, BPF_REG_AX, 0);
7 BPF_ALU64_IMM(BPF_ARSH, BPF_REG_AX, 63);
8 BPF_ALU64_REG(BPF_AND, BPF_REG_AX, off_reg);
```

Listing 2.7: eBPF pointer sanitization

After we know the offset is positive, we get the maximum value this offset can have in the current execution context. The static analysis of the verifier calculates this value, stores it in `aux->alu_limit`. Now we subtract the offset from the maximum

value. Notice that the value is always positive if the offset is below or equal to the limit; next, we *or* the two registers. Here the value is above 0 if everything is as expected. In line 6, we negate the value. A positive number has not set the bit at position 64, and a negative number has the most significant bit (msb) equal to 1. We extend the most significant bit to the whole number in line 7. If the access is in bounds, i.e., the offset is less or equal `aux->alu_limit`, we logical *and* with the number -1. If the offset was bigger than the defined limit, we *and* with 0, so the register `BPF_REG_AX` is always 0. Thus we add 0 to our pointer in the speculative domain if the offset is out-of-bounds. How this looks for real values can be seen in B.2 and B.3.

Speculative Store Bypass

To prevent the speculative store bypass attack a preventive store of 0 has been added if the stack slot was used before in the current execution context. Jann Horn also discovered the issue [5]. However, he was not mentioned in the patch [26]. In Listing 2.8 is an example of how such an attack looks. Here r8 and r6 point to the same stack address. The store in line 4 is skipped since the address r8 is unknown due to the slow dereference of r7. Because r8 is unknown, the CPU speculates there is no RAW dependency between instruction 4 and 5 and continues execution with instruction 5.

```
1 (bf) r3 = r10
2 (07) r3 += -216
3 (79) r8 = *(u64 *)(r7 +0) // slow read
4 (7b) *(u64 *)(r8 +0) = r3 // this store becomes slow due to r8
5 (79) r1 = *(u64 *)(r6 +0) // cpu speculatively executes this load
6 (71) r2 = *(u8 *)(r1 +0) // speculatively arbitrary 'load byte'
```

Listing 2.8: eBPF Spectre v4 example

To prevent this attack, before instruction four there is injected a preventive store of 0 on the stack address of r8. If we would use r8 to store the 0, this could be ignored as well, so instead, we are using the static offset from the frame pointer r10, which is guaranteed to be fast since it is a static register with minimal latency. The fast load is guaranteed because BPF programs run without preemption, and thus no context switch is possible and no rescheduling on a new CPU core.

```
1 0: (7a) *(u64 *)(r10 -72) = 0 // sanitize instruction
```

Listing 2.9: eBPF Spectre v4 countermeasure

After the instruction in Listing 2.9 is inserted between instructions 3 and 4, the attack is not exploitable anymore.

Branch Target Injection

Branch target injection, also known as Spectre v2, protection in eBPF is only applied for one special case. Since Linux version 4.2 [27] tail calls are available. It is possible to chain executions of up to 33 BPF programs with tail calls. It is comparable with the `execve` system call; once a program executes a tail call, it does not return anymore, and the called program replaces the old one and uses the same stack memory region. Tail calls are implemented using jump instructions. So instead of doing a call, we jump to the target address of the called program. The target address is loaded from a special map, `BPF_MAP_TYPE_PROG_ARRAY`.

The relative jump could introduce a branch target injection attack, where the register where the jump should go speculatively contain a wrong value. The standard retpoline mitigation has been inserted to prevent this behavior for affected processors [28]. It ensures that all load operations are performed before the code after the `lfence` instruction is executed and that no speculation happens. The CPU cannot speculate about the jump location anymore since the program stalls until the exact location is resolved. Retpolines are pretty expensive. The verifier is also able to rewrite tail calls if the program which gets called is static at compile-time to prevent retpolines [29].

2.2.7 Program Attachment

The `BPF_PROG_LOAD` command only loads the program into the kernel. It lies in the kernel, somehow like a new function that can get executed. The program can be executed using the `BPF_PROG_TESTRUN` command, but this is only for testing purposes and not used in production. The usual way a program is started is the event it is attached to gets triggered. The events on which a program can be attached to are program-specific since the program type determines which data structure resides in register r1 when the program is started.

The program type `SOCKET_FILTER` can be attached to sockets with the userspace function `setsockopt` and the corresponding attachment parameter `SO_ATTACH_BPF`. After the socket is attached to the BPF program, the program runs when the socket processes a package that calls `sock_queue_rcv_skb` during the package handling procedure within the kernel [30]. The program can trim the package or drop it.

kprobes, tracepoints, and perf programs get attached trough the `ioctl` call, which gets the arguments seen in Listing 2.10.

```
1 ioctl(perf_fd, PERF_EVENT_IOC_SET_BPF, bpf_prog_fd);
2 ioctl(perf_fd, PERF_EVENT_IOC_ENABLE, 0);
```

Listing 2.10: `ioctl` eBPF attachment system calls

2.2.8 BPF Type Format

BTF is used for encoding data structures such as structs for use within eBPF. It was extended to provide also information for functions, line info annotations, and global variables.

`LLVM` can generate the BTF information, and when the program gets loaded using the corresponding `libbpf` helpers, it is possible to obtain the annotated lines with bpftool. When using BPF for tracing, a problem that often occurs is that some structs in the kernel do not appear in the kernel headers. Thus, it is not possible to navigate through that struct. BTF can change it by providing offsets for these structs while being more lightweight than `debuginfo`. It is only allowed for a privileged user to load a new BTF into the kernel.

2.3 Fuzzing

To search automatically for bugs, we want to use an automatic bug detection technique. Over the past few years, *coverage guided fuzzing* has been the most successful attempt in that direction. Modern implementation work like reinforcement learning algorithms. They execute the program with a specific input and obtain the path coverage the input has caused. The goal is to increase the path coverage as much as possible. The program mutates the input in order to achieve higher coverage. If the mutated input achieved a higher coverage or uncovered a new path, it is stored in a queue for further mutations. If not, the input gets rejected.

This dramatically simplifies the construction of such fuzzers, but more information is not necessary to understand what follows since we do not construct a fuzzer from the bottom up. Preferably we use proven state-of-the-art coverage guided fuzzers. The first fuzzer we use is syzkaller, which has uncovered over 300 bugs in the Linux kernel. However, not all bugs are security-related, and only a few got a CVE assigned [31]. Syzkaller is already capable of fuzzing eBPF. To have something new, which we could compare against, we also use kAFL, which was developed by Sergej Schumilo [32]. This fuzzer has already proven that it is capable of uncovering Linux bugs and provides good performance. Later it was extended to achieve higher path coverage with mutated inputs and shows even better results [33][34][35].

We are searching for memory-related issues using the in-kernel sanitizer KASAN [36]. It can detect use-after-free vulnerabilities or out-of-bounds access to certain memory regions. The other sanitizer we use is UBSAN [37], which is used to detect undefined behavior, for example, a null pointer usage or integer over and underflows.

3 Methods

We use different methods to cover as many aspects of software security as possible. First, fuzzing the software is the best we can do to find bugs without much a priori information. Secondly, manual analysis is used to find more complex vulnerabilities.

3.1 Fuzzing

Fuzzing is an excellent approach to analyze the security of a given software. If it is correctly used, many programming flaws can be detected. Thus we use a fuzzer to find bugs in the BPF subsystem. Since there are many options for the BPF system call, many agents can be written to test these software parts specifically.

3.1.1 kAFL

There are not many well-suited kernel fuzzers. The fuzzer with the most success over time is syzkaller, which a google engineer developed. For the thesis, three options were under consideration.

1. Writing a kernel fuzzer from scratch using gcov does not scale and would exceed the limit for this thesis.
2. syzkaller was already able to fuzz the BPF ecosystem, and the only way the fuzzer could have been extended is through advanced syntax definitions for eBPF.
3. The last and most promising option is to use kAFL, which was developed by Schumilo [32] because this fuzzer has not been tested on eBPF specifically; thus, the probability that the fuzzer will find bugs is high.

kAFL is only usable on Intel CPUs, which support the Intel-PT extension. These are all Core I series processors from the Skylake generation onwards. The fuzzer works by tracing branches in the processor using hardware features. Independence from software is achieved, so even BlackBox programs can be traced using this technique since no software instrumentation is necessary. The fuzzer works by setting up a modified QEMU version in which any OS can run. The virtual addresses of the

tracing range inside the guest need to be obtained and passed to the fuzzer. The fuzzer is configured to receive branch information from within the guest from the specified address range. The fuzzer also overwrites particular error handle addresses, for example, the panic hander. This is used that in case a crashing input was found, the hypervisor gets notified about the event. It is done by patching the handler address with a hypercall, and transmit control to the QEMU instance, which can read out information and pass them to the fuzzer.

As a fuzzing setup, we use Linux kernel 5.11.4 with all BPF options enabled, which are discussed in Section 2.2.2. The most likely errors which occur in software are memory corruption errors. These not necessarily led to a kernel panic, for example, and out-of-bounds read does not alter the system state. Nonetheless, these errors must be detected. Therefore, the kernel was compiled with KASAN instrumentation, a live instrumentation technique that checks for memory corruption errors. If an error occurs, a KASAN event is triggered, and the execution context is transferred to the KASAN handler. Since the fuzzer needs to be aware of this event, the handler address is also patched, just like the panic address. For further extension, other Linux sanitization techniques are used, such as UBSAN. Currently, kAFL only supports one of those two sanitizers because only one hypercall is defined for such a sanitizer. However, this could be easily overcome.

The first test setup on an Intel Core `i5 8250u` CPU was unsuccessful because the Intel PT decoder, which decodes the bitstream that the hardware provides into useful coverage guidance information, was sometimes unable to process the received packets. The reason is that a buffer was full, and the interrupt to clear it was not correctly recognized by the system. So no information could be written to that buffer, and the decoder complained. Since this is a low-level problem and nasty to fix, we decided to use a system from the university. The setup consists of 32 gigabyte RAM and an Intel Core `i7 6700` CPU. With this setup, everything works as expected using `Ubuntu 18-04 LTS` as the host system. The guest system we fuzz consists of the Linux 5.11.4 kernel and a busybox instance as an init process.

3.1.2 Syzkaller

To compare the results of kAFL against another fuzzer, we also run syzkaller on another machine. While the fuzzers are not directly comparable, since syzkaller uses syntax definitions, it is the best fuzzer currently available for Linux fuzzing. We tuned the syzkaller config that only eBPF related system calls get executed. The kernel image was compiled with every sanitization technique which Linux offers enabled. All eBPF options discussed in Section 2.2.2 are enabled. The machine syzkaller runs on has an Intel `Xeon E3-1231 v3` as CPU. We also use the results which syzkaller has found in the past on machines with much more computing power.

3.2 Manual Inspection

While the fuzzers are running, we use the time to also search manually for bugs. There are two research fields where we could look closer for vulnerabilities. The first one is to search for logical bugs, where the verifier wrongly tracks a variable and the second one is to search for Spectre-related bugs. The last approach seemed to be very successful since Piotr Krysiuk has already found such vulnerabilities. However, even the first type of bug remains inside the verifier [7].

We decide to search for Spectre vulnerabilities and find a speculative execution path where the verifier has not sanitized the code correctly. Some fuzzers are designed to find such a path, but extending them to work on eBPF is out of scope and not worth the effort since we would need an entirely new concept due to the specialty of eBPF programs. Spectre vulnerabilities are extremely hard to discover, and lots of creativity and knowledge is necessary to design proof-of-concept codes that use the speculative domain. Since we can supply the code that gets executed in the kernel, we can search for all Spectre vulnerabilities; these are speculative bounds check bypass or speculative store bypass. The other issues which are Spectre related, e.g., L1TF are using hyperthreading to get data from the other thread and cross privilege boundaries. These are not trivial to exploit, and we leave the scope of BPF programs, and we would also need to consider other Linux kernel parts. In that case, we would exceed the limit for this thesis. However, for future work, it might be interesting to search for such bugs as well.

The verifier itself returns a log of the program verification run. Depending on the system call log level, it contains the range tracking for all variables and the stack after every executed code line. Also, if the program gets rejected, the reason for it is provided in the log. This information is used to craft inputs that potentially bypass the Spectre mitigations. If a potential bypass was discovered, we try to exploit it using the code, which is explained in Chapter 4. After the successful exploitation, a report was sent to Daniel and Alexei to work on a fix for it.

4 Implementation

The implementation part consists of the code crafted for the fuzzer and the proof-of-concept code for the Spectre vulnerabilities. The latter code is based on Jann Horn's proof-of-concept codes for the Spectre v1 and v4 vulnerabilities within the eBPF ecosystem. The code was extended to work on other variants of the vulnerabilities, but the central concept of increasing the latency of a load remains the same.

4.1 kAFL Agents

kAFL was used for kernel fuzzing before but not for system calls specifically. The main field of use was kernel module fuzzing. Thus, the software for determining the fuzzing range was only capable of processing modules. The kernel needs to be compiled with `debug_info` to obtain the addresses of all kernel functions. We can then access the location of all functions from the kernel image. Alternatively, `/proc/kallsyms` could be used. However, not all symbols appear here; some may be inlined and thus ignored in kallsyms. To trace the BPF system call, we first need to know which files and functions in these files could be called if the BPF system call is invoked. Especially these are

1. `/kernel/bpf/verifier.c` All function calls in here are used for the verification process of an `BPF_PROG_LOAD` system call.
2. `/kernel/bpf/core.c` The interpreter resides in this function, and it is not commonly used anymore since JIT compiling is much faster.
3. `/kernel/bpf/btf.c` Functions that are used to process the `BPF_BTF_LOAD` call are defined in this file.
4. `/arch/x86/net/bpf_jit_comp.c` Functions in this file are used to JIT the BPF bytecode into machine code. This file is specifically for x86. However, the JIT compiler for other functions resides in their corresponding arch directory.

Several other files are also used and can be found in the directory `/kernel/bpf/`. These have all self-explaining names and also need to be traced. Luckily if KASLR is turned off, the function in these files resides one after another in the memory, and we only have to define one tracing range. Only the JIT compiler is in another tree for legacy reasons. cBPF was used only for network filtering. Thus, the JIT compiler

is in the `net-tree`. Nevertheless, for our fuzzing purpose, we do not need to trace the JIT compiler code. We are not specifically hunting for logical flaws, and the JIT compiler is a big switch-case statement. Instead, we are searching for memory corruption issues. Thus, we do not care if the program gets accepted or rejected, and the JIT compiler itself consists of a big switch-statement block and is unlikely to contain any memory corruption issues. To further extend the fuzzer, it could be possible to search also specifically for JIT compiler bugs. However, path coverage is not the optimal metric in this case.

The agents themselves call the BPF system call with the corresponding arguments. These are command-specific structs that are grouped in a union `bpf_attr`. Some arguments of these structs are statically set, and some are set by the fuzzer. The static arguments have little impact on the path coverage. For example, the kernel version in the `BPF_PROG_LOAD` call is only used to verify if the program can run since some kprobes are not accessible from older kernels. If the kernel version does not match, the program exits with the corresponding error message. Thus, we can set the number to a static value so that it can be ignored. The decisions to leave some fields static and some to be instrumented by the fuzzer are explained in Chapter 5. The binary used for program loading needs unrestricted access to `/proc/kallsyms` to obtain the panic and KASAN handler addresses and submit these to the hypervisor. So after a panic or KASAN event, the hypervisor can take control over the execution and submit information to the fuzzer. For this, the agent has to be executed as root. However, the goal is to find security vulnerabilities, and we have to drop the privileges so the BPF system call can be executed as a regular user without any additional permissions.

4.2 Spectre Proof-of-Concept

While it is relatively easy to discover a side-channel attack, successful exploitation is complicated. If we are in a constrained sandbox-like eBPF, it is even more challenging to use the side-channel, and much system knowledge is needed to exfiltrate data through the side-channel successfully.

Since eBPF is a sandbox, we have to work around some restrictions. The first problem which occurs is the absence of the `clflush` instruction which is used in many proof-of-concept codes for the Spectre v1 vulnerability. The instruction is used for flushing a cache line that usually contains a pointer to the value that should be loaded slowly. If the value does not reside in the cache anymore, accessing it is **slow**. If such a value is loaded, we refer to it as a **slow load**. These slowly loaded values are the key for using side-channel attacks in the speculative domain. To achieve a slow load in eBPF, we use the cache coherency within modern processors.

The proof-of-concept code is based on Jann Horn's [4][5] code which was developed back in 2018. Understanding this code is essential since the exploitation of the vulnerabilities found relies on its basic idea. No other Spectre proof-of-concept code is attacking the Spectre vulnerabilities inside the Linux kernel, and many other researchers also use this as a base for their exploitation [38].

eBPF allows us to call pre-defined functions and a restricted set of these functions is also accessible from userspace. One of these functions is `bpf_ktime_get_ns` which was intentionally used to timestamp the receive of network packages. However, this function could also be used to measure the time of a pointer dereference. So we can use it as a timer for our Spectre attack. The resolution of the timer is precise enough to measure the difference between an L1 and L3 hit.

To achieve a high latency load, we use maps on two different physical cores, core **A** and core **B**. Consider the example that core **B** changed a value of the shared map, and core **A** wants to access it. The value resides in the L1 cache of the physical core **B**, but core **A** has not got direct access to that cache. There have been invented multiple cache coherency protocols to synchronize those caches. The MESI protocol was already introduced in Section 2.1.4. It has been extended by certain other flags and is used in modern processors to ensure cache coherency. However, most desktop CPU architectures use the L3 cache for synchronization.

If core **B** modifies the value, the L1/L2 cache line with the same value is marked as **invalid** in the L1/L2 cache of core **A**. Depending on the processor model, the L3 cache already contains the updated value from core **B** or knows in which L1/L2 the modified value is and initiates a cache-to-cache transfer if core **A** want to access the value. It is around 15 times slower than an L1 hit and gives core **A** time to execute other instructions out-of-order speculatively. We use this technique as an alternative for the `cflush` instruction since the latency for an L3 load is high enough to execute a Spectre gadget in the speculative domain.

4.2.1 Bounds Check Bypass Implementation

We use the values inside the L3 cache for two purposes in the proof-of-concept code. For a slow load and to be able to differentiate which values are loaded in the speculative domain. The code consists of three BPF programs that get loaded into the kernel. The code for the speculative store bypass varies slightly. It is covered in the next section.

1. The first program is used to trigger the side-channel attack and one branch in here should get executed speculatively.

2. The second program runs on the same physical core as the first and measures the latency to access a value inside the **data map**.

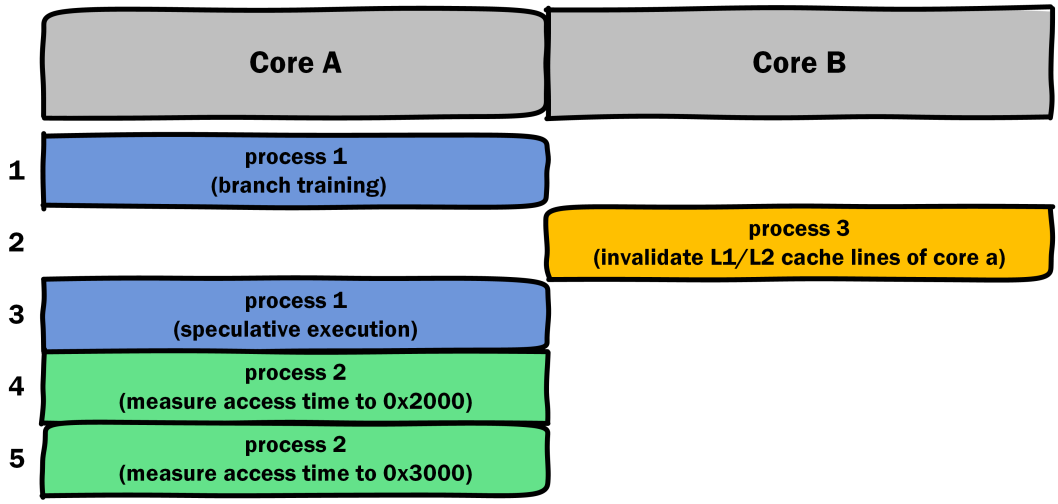


Figure 4.1: High-level execution flow of the eBPF Spectre v1 exploit code

3. The third program runs on a different physical core and is used to increase the load latency for specific array accesses in the first program. It moves the **data map** elements into the L3 cache for core **A**. So it is possible to distinguish which elements got accessed in the speculative domain in the first program.

We have one **control map** for the first program and a **data map**, which is shared between the three eBPF programs.

We leak the kernel data bitwise since it is faster as we do not have to move 256 array elements into the L3 cache and measure the access latency for all of them. It is also more error resistance since we only have two latencies to compare and less noise to deal with.

Technically the proof-of-concept code uses `pthread` to let functions run on another core. The executing core is changed by setting the scheduler affinity to the core of choice. With it, we ensure that both processes do not run on the same physical core.

The execution mechanism can be seen in Figure 4.1. In the beginning, we trigger the first program with the control index set to 0. This step fulfills the purpose of miss training the branch predictor. Step 2 is used to invalidate cache lines of core **A**. These lines are

1. **data map** index at offset `0x3000`, is loaded back in the cache, if we speculatively loaded the bit 1

2. **data map** index at offset `0x2000`, is loaded back in the cache, if we speculatively loaded the bit 0

3. **data map** index at offset `0x1000`, is loaded back every time, since the value is used, to achieve a slow load.

In line 3, the leaker program is executed. We obtain the side-channel information and load one of the **data map** values in the cache. Either we speculatively access the address `0x2000` if the leaked bit is a 0 or `0x3000` if it is a 1. On core A we measure the access times to both array elements and check which access was faster. Both entries in the L1/L2 cache of core **A** were invalid due to the access from core **B** in line 2. Only the loaded one gets back into the L1 cache in step 3. The element with the lowest access time determines the speculatively loaded bit.

The difference between the array elements is exactly `0x1000`, which translates to 4KiB, which is the usual page size in Linux. The reason is that even if a cache line does not consist of 4KiB, the whole page may be loaded anyway in the cache due to speculation that the data may be needed in the future.

4.2.2 Store Bypass Implementation

The speculative store bypass proof-of-concept code is slightly different. We have multiple values loaded into the cache when the instructions get re-executed in the non-speculative domain.

The methods to load the map values into the L3 cache are the same, only the leak procedure slightly varies. The three different BPF programs are also conceptually the same as introduced in the previous section. Programm two and three are identical, whereas the first program differs since we are using another attack vector. Here we

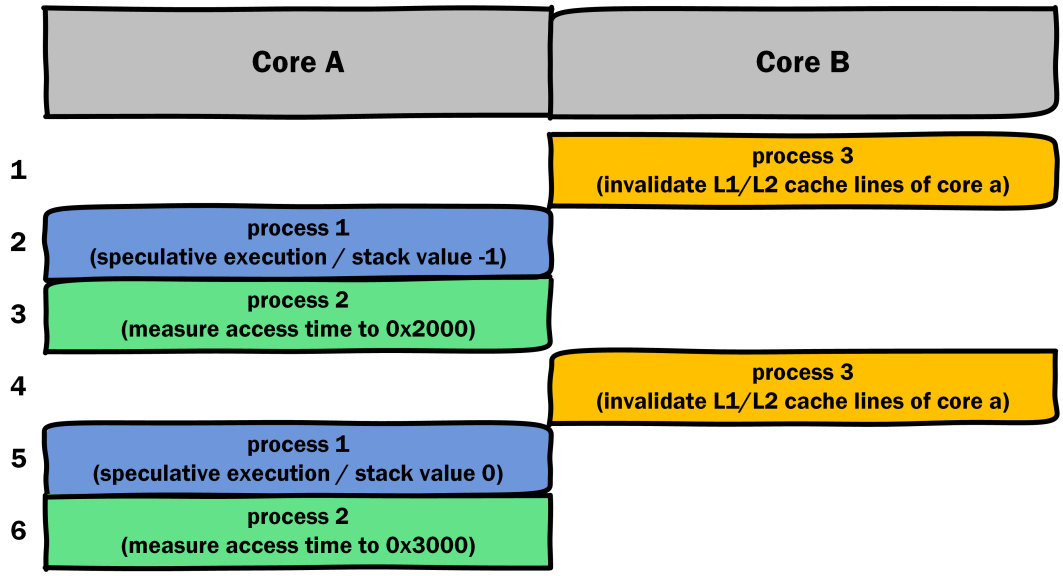


Figure 4.2: High-level execution flow of the eBPF Spectre v4 exploit code

first set the address we want to read as the index value in our map. Then we load the map values in the L3 cache, so accessing them is slow. In the next step, we trigger process 1 with the parameter `-1`. As we are in the `two's complement` this translate into the value `0xffffffff`. The address of the pointer, which assignment got bypassed, points to that value. Meaning if the CPU detects the misprediction, we re-execute all instructions and load the value at the offset `0x3000` into the cache. In lines 2 and 3, we only obtain information about the access time to the index `0x2000` and ignore index `0x3000`, since it is always in the cache. Line 5 to 6 repeats the process but vice versa, meaning the pointer points to `0x0` and we only measure the access latency to the array element `0x3000`. In the last step, we compare those two values, and if one value is below a pre-defined threshold, we obtain information about the bit at the index we set.

The fundamental difference between the speculative bounds check bypass is that the probability that a speculative store bypass happens in a general case is below 0.1%. This probability also decreases in some cases due to the restrictions which apply in eBPF. Mainly the absence of the `clflush`. Nonetheless, we have worked around the restrictions and designed a proof-of-concept with a trigger rate of around 5% in some cases on some CPUs. The code ran successfully on an Intel Core `i5-8250u`. However, this method has a trigger rate of 0% on an `Xeon E3-1231 v3`.

Currently, we are unsure what the root cause is, and further research is needed to pin it down definitely. However, as we can see, things become complicated and highly architecture-dependent when we leave the scope of the easy to exploit speculative bounds check bypass vulnerabilities.

5 Fuzzing Results

This section contains the fuzzing results. First, we evaluate the path coverage that kAFL achieved. Afterward, we analyze the finding and discuss why path coverage is not the best metric for eBPF fuzzing and how to improve eBPF fuzzing.

5.1 Path Coverage

kAFL collects data of the fuzzing run and makes it user-accessible. It makes it possible to compare fuzzing runs and evaluate how many different paths were discovered. However, the path count should be taken with care because it is not equivalent to code coverage. Nonetheless, it is the best metric that kAFL offers for comparison. We only evaluate three different system call arguments because many of the arguments do not have different paths that can be run through, or the input dependence on the path happens in another region of the kernel. Thus these are the only ones, which kAFL can instrument well.

5.1.1 Program Loading

The system call command with the highest path coverage was the `BPF_PROG_LOAD` call. Here over 30.000 paths were discovered by kAFL. As we can see in Figure 5.1, the rise is nearly linear after we hit a breakpoint. The hypothesis is that we have a large for-loop in the verification procedure, which only exits with a break instruction. Within the loop, many function calls are performed. They generate many new paths in every execution and potentially wrongly guide the fuzzer.

The agent obtained the information from the fuzzer and parsed them as a program argument; the other parameters were static since they only slightly influence the path coverage. How the parameters are exactly set is shown in Listing 5.1.

```
1  char verifier_log[100000];
2  union bpf_attr create_prog_attrs = {
3    .prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
4    .insn_cnt = 0,
5    .insns = 0,
```

```
6     .license = (unsigned long)"GPL",
7     .log_level = 4,
8     .log_size = sizeof(verifier_log),
9     .log_buf = (uint64_t)verifier_log
10  };
11  [...]
12    create_prog_attrs.insn_cnt = (payload_buffer->size)>>3;
13    create_prog_attrs.insns = payload_buffer->data;
```

Listing 5.1: `BPF_PROG_LOAD` agent

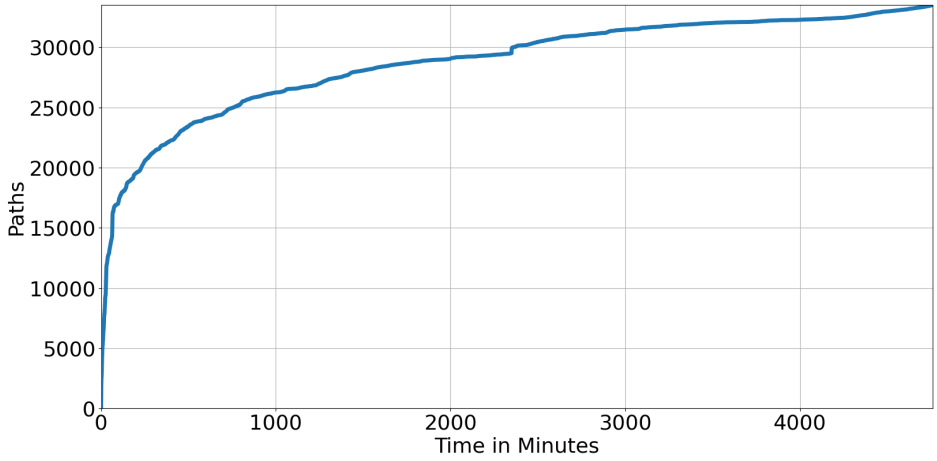


Figure 5.1: Path coverage `BPF_PROG_LOAD`

In the beginning, many KASAN events were detected. Unfortunately, these were false positives. The reason for it is that the wrong function was hooked. When loading a BPF program, kernel memory is written under certain conditions to patch a BPF program. Since accessing executable kernel memory that way could be a security risk the `kasan_report` function is triggered, which collects data. During the function's execution, the kernel notices that this access was valid and thus throws the event away. To prevent the false positives, we hook into the function `end_report` instead, which only triggers right before an event is written into the kernel log buffer.

The system call command was also fuzzed as root. However, that resulted in many timeouts. Since back edges are allowed for root programs, the fuzzer mainly relies on them since every iteration must be checked and thus provides additional path coverage because a new path is executed with every iteration. Fuzzing with higher privileges was not possible because of this issue. Some programs even needed around 2 minutes to get verified. Nevertheless, fuzzing BPF as root is not as interesting as the

unprivileged case because we search for security-relevant bugs.

5.1.2 Map Creation

The other system call parameter, which got a high path coverage, is `BPF_MAP_CREATE`. For this argument, the information, which the fuzzer provides is directly interpreted as `bpf_attr` as shown in Listing 5.2.

```
1 int ret = syscall(__NR_bpf, BPF_MAP_CREATE , payload_buffer->data,
      payload_buffer->size);
```

Listing 5.2: `BPF_MAP_CREATE` agent

As we can see in Listing 2.5 no pointer arguments are necessary for loading. Only four integer values are parsed. The struct can be longer using options that are not shown in the listing, but they are also only integer constants. The full and up-to-date definition of the struct can be obtained from `/include/uapi/Linux/bpf.h`. Since the user-API is stable, the actual file might change, but the folder will be the same. The path exploration over time can be observed from Figure 5.2. As we can

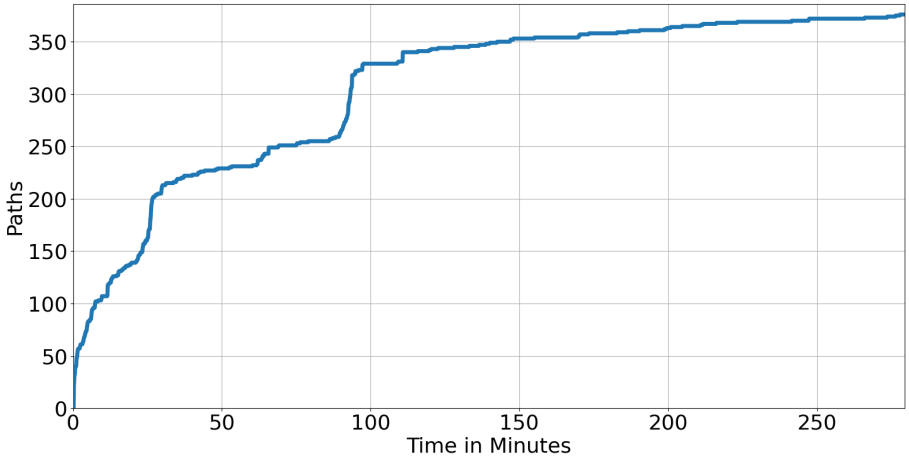


Figure 5.2: Path coverage `BPF_MAP_CREATE`

see, at certain points, the path coverage seems to stagnate, and then it makes a big jump upwards. The reason for it is that in this case, the fuzzer has detected a new map type, which has a different path in the kernel, e.g., the ring buffer. We rerun the agent different times, and we could always observe the same result. Unfortunately, we got many false positives.

5.1.3 BTF Loading

Initially, the BTF system call seems promising for fuzzing since many descriptions can be dynamically loaded into the kernel. However, this is a privileged system call option, so we either need `CAP_BPF` or `CAP_SYS_ADMIN` to execute it. It is not security-related in the first place since not many users have `CAP_BPF` in most distributions, but escalating the privileges from this capability to root could be a security-related bug, as well. Furthermore, the path coverage on this system call was not as good as expected. We only observed five different execution paths even after we ran the agent for two days. The reason for that could be that the BTF only gets loaded into the kernel, similar to the program. Therefore, we need to use it before a malicious BTF object could harm the system. However, this is only a hypothesis, which has to be confirmed in further research.

5.2 Findings

While fuzzing, we observed many false-positive results. Unfortunately, we did not uncover new critical vulnerabilities. The false positives were distribution-related issues, which root cause is the lacking use of cgroup restrictions. On most distributions, no cgroup related memory restrictions are applied on a per-user basis. That means that a normal user can occupy the whole memory of the system. In such a case, the out-of-memory killer kicks in and starts to kill processes, which belong to the same group. If the memory limit in the cgroup file `/sys/fs/cgroup/user.slice/memory.max` is set to `max` this could led to a race condition, which could result in a kernel panic with the error message *deadlocked on memory*. The developers were already aware of such memory panics, and they are not unique to eBPF. The responsible persons for fixing it are either the memory subsystem maintainers, who should fix the race condition, or the distribution developers, who should set up cgroups for the user with a more restricted configuration.

Until now, it is possible to deadlock the kernel under certain conditions with the creation of `BPF_MAP_TYPE_PERCPU_ARRAY`. The other map types are not vulnerable. We assume this is a memory allocation problem, but the Linux kernel's memory subsystem is far too complex for debugging it as someone with little kernel experience.

5.3 Evaluation

Regrettably, no security-relevant bugs were found. There are several reasons for it. The main component of eBPF is program loading. Thus, we discuss this in-depth. Some of the issues also apply to the other system call option.

1. We have not the best-suited sanitizer to uncover logical bugs.
2. We use the wrong metric; logical bugs are not influenced by path coverage of the verifier.
3. The kernel code is nearly free from trivial bugs.

The sanitizer KASAN and UBSAN are not the best if it comes to eBPF. Since the programs are JIT-compiled, they cannot instrument the JITed code. Also, for a KASAN event to be triggered within an eBPF program, many conditions must be met. The fuzzer may have already generated a program where the verifier wrongly tracks a range. This range now has to be used to access the memory out-of-bounds. So the wrongly tracked value has to be in bounds and the actual value out of bounds. It is improbable to happen. So another technique has to be applied to fuzz more efficient.

Also, path coverage is not the best metric when it comes to logical bugs within eBPF. The verifier only checks the program, which gets loaded into memory. Maybe two paths for two different instructions are enough to overcome the verifier range tracking system. However, this may not be noticed by the fuzzer since it is trying to maximize the path coverage. There is little correlation between logical eBPF bugs and path coverage. The fuzzer does not use any information, which is essential to find logical bugs. Nonetheless, path coverage also has its good sides. Modern fuzzers work like reinforcement learning algorithms. We do not give kAFL any information about the syntax nor semantics of the eBPF assembly langue. However, the fuzzer could generate syntactically correct programs accepted by the verifier. Black box fuzzing cannot achieve this without a syntax definition. kAFL is learning this definition through the path coverage information provided by Intel-PT.

Another reason that we found no critical bugs is that the kernel code is usually solid. eBPF was introduced with Linux kernel 3.18. Syzkaller can fuzz eBPF, and many people have already looked at the source code. Thus uncovering hard-to-find bugs is unlikely to happen if we do not provide additional a priori information as Scannell did [8]. Also, syzkaller has only found five vulnerabilities in the eBPF subsystem, the most in the early stage, right after eBPF got upstream. One vulnerability was also not considered as a real issue [39]. In total, only one vulnerability got a CVE assigned.

To summarize, fuzzing the `BPF_PROG_LOAD` argument of eBPF with coverage guided fuzzers was not as successful in the past for uncovering security-related vulnerabilities. We show that coverage-guided fuzzing alone is not ideal for eBPF, and we need manual techniques to discover certain types of bugs. These are precisely logical flaws in the speculative and non-speculative domain. Nonetheless, Anatoly Trosinenko [40] and Simon Scannell [8] have developed fuzzers, which uncovered issues within the verifier. They are specially designed for the eBPF ecosystem and bring a priori knowledge heavily in the fuzzer.

6 Spectre Vulnerabilities

We found at least two vulnerabilities with a CVE assigned during the research and one potential bypass of the speculative bound bypass mitigation with a reserved CVE. In addition, we introduce another speculative bypass, which was discovered by another researcher working on the same topic. For each vulnerability also the essential part of the proof-of-concept is covered.

6.1 CVE-2021-29155

This vulnerability is a classical speculative bounds check bypass. The root cause of it was a flaw in the implementation of the pointer masking scheme.

6.1.1 Attack

We discovered an issue with the Spectre mitigations inside the verifier. Since the Spectre vulnerabilities have been publicly disclosed in early 2018, the eBPF ecosystem has many patches applied to harden it against these attacks. The pointer masking scheme was already introduced in Chapter 2. The implementation of this technique was vulnerable against attacks until Linux version 5.11.16. After discovering the vulnerability and reading some old posts, the main idea behind this attack vector was already mentioned by Manfred Paul in another context in his blogpost [6]. The issue was that the pointer masking was not tightened to the maximum value the offset variable can have.

The abuse of this vulnerability is simple as it is a straight forward Spectre v1 issue also known as speculative bounds check bypass. Since all prequisites necessary for understanding the vulnerability are covered in previous sections we only cover the actual error in here.

```
1 /* show r9 is in range for the static verifier */
2 (c5) if r9 s< 0x4fff goto pc+2
3 (b7) r0 = 0
4 (95) exit
5 (65) if r9 s> 0x0 goto pc+2
6 (b7) r0 = 0
7 (95) exit
```

```
8  /* offset masking */
9  (b4) w11 = 20479
10 (1f) r11 -= r9
11 (4f) r11 |= r9
12 (87) r11 = -r11
13 (c7) r11 s>>= 63
14 (5f) r11 &= r9
15 /* add the scalar value to r4 */
16 (0f) r4 += r11
17 /* load leak parameter into r1/r2 */
18 (79) r1 = *(u64 *)(r7 +0)
19 (79) r2 = *(u64 *)(r7 +8)
20 (57) r1 &= 15
21 /* slowly load r3 */
22 (79) r3 = *(u64 *)(r0 +4608)
23 (57) r3 &= 1
24 (57) r3 &= 2
25 /* spculative branch */
26 (2d) if r2 > r3 goto pc+19
27 /* insufficient masking */
28 (b4) w11 = 20478
29 (1f) r11 -= r2
30 (4f) r11 |= r2
31 (87) r11 = -r11
32 (c7) r11 s>>= 63
33 (5f) r11 &= r2
34 (0f) r4 += r11
35 /* speculative OOB load */
36 (71) r4 = *(u8 *)(r4 +0)
37 /* exfiltration */
38 [...]
```

Listing 6.1: CVE-2021-29155 proof-of-concept code

The masking scheme was that a known offset range was sanitized using the maximum value which we could add to the pointer. Even if we add cumulative multiple scalar values to the pointer, the masking value only decreases by the minimum value the offset can hold for each addition. The static verifier would still reject values that could potentially be out-of-bounds in the non-speculative domain, but the masking value was not set accordingly for the speculative domain.

The attack is shown Listing 6.1. First, we load a scalar value from a map inside r9. The register content is completely unknown until both jumps are executed (lines 2-7). Now the verifier knows that the value is between `0x1` and `0x4ffe`. We add the value r9 to our pointer r4 (lines 9-14). Since r4 has a maximum size of `0x5000` the maximum

we can add is `0x4fff` and that is the masking value in line 9. However, the pointer masking gets only decreased by 1. In line 26, we make sure that the next offset we add to our pointer is 0 in the non-speculative domain, so the verifier assumes everything is safe, but the masking scheme for the case is not tight enough. r4 can contain the offset `0x4ffe`, and the masking scheme allows this offset to be added to the index in the speculative domain. Index `0x9ffc` is then speculatively accessed. Thus, we can read approximately the map size out-of-bounds.

The issue was also discovered by Piotr Krysiuk [41].

6.1.2 Fix

The patch was relatively simple but required many previous changes to the verifier logic [41]. Instead of decreasing the masking value by the minimum value which a dynamic offset can hold, the masking value is calculated by the maximum value the scalar added to the pointer can contain. The information was already calculated by the verifier when checking the non-speculative domain. The patch series makes this information usable for pointer sanitization after the concerned branch statement. It means that the masking value for the verifier would be decreased to 0 in line 28 since we know that r2 has the value 0 if we do not take the jump in line 26.

To summarize, the above Spectre attack is not possible anymore since the masking offset is tightened and adjusted according to the potential value the register can contain; instead of relying on the maximum value, we can potentially add to the pointer.

6.2 CVE-2021-34556 Reserved

This vulnerability is a theoretical speculative store bypass, which we could only exploit with a patched kernel. A more skilled attacker could also exploit this vulnerability with an unpatched kernel.

6.2.1 Attack

We bypassed the Spectre v4 mitigations in the current Linux version (5.14). At the time we finished the thesis, there is no patch publicly available. The problem is that the preventive store of 0 only occurs if the stack slot was previously used in the current execution context. However, it is experimentally verified that particular stack slots preserve their values through runs. It follows that the uninitialized stack could contain data written to the stack in a previous execution of the same program.

The verifier needs to be patched to read data from an uninitialized stack to detect this behavior. Without a kernel patch, we are not able to read the stack even as root. With the patched kernel, we write a program that, depending on the data we set in the control map, first writes an address to the stack and, in the second run, reads uninitialized stack data and checks whether the data has changed. We did this 1000 times, and a percentage score of the probability that the stack data at a specific slot gets overwritten was calculated. For example, empirically, stack slot offset −424 was only overwritten in 2 out of 1000 cases at maximum, and it was consistent over several program executions.

```
Index 424 Got # 0 erros which is a rate of 0.00%
```

Listing 6.2: Rewrite probability stack offset `-424`

Other stack addresses are used every time. It is assumed that another kernel process uses the stack address space if the BPF program is not running. Nevertheless, not every allocated space is overwritten.

For demonstration purposes, we use a patched kernel. The patch allowed us to use the `clflush` instruction within the eBPF context. It was necessary to get a slow stack load. The `clflush` instruction is used for flushing a cache line. It ensures that data, which a pointer points to, get out of the cache. If the pointer is dereferenced the next time, the data must be pulled out of the main memory, which is slow.

```
(bf) r1 = r7
/* flush the cache line of pointer r7 */
(85) call bpf_clflush_mfence#137744
(bf) r3 = r10
(07) r3 += -216
/* skip the slow derefencre of r7 */
(79) r8 = *(u64 *)(r7 +0)
(7b) *(u64 *)(r8 +0) = r3
/* speculatively access r6 which points
   to the same address as r8 */
(79) r1 = *(u64 *)(r6 +0)
(71) r2 = *(u8 *)(r1 +0)
[...]
```

Listing 6.3: CVE-2021-34556 assisted proof-of-concept code

Listing 6.3 shows the eBPF assembly of the assisted exploit. We have patched the kernel and defined another eBPF function, which allows us to flush the cache line which resides in r1. As we can see in the code, we are repressing r7 out of the cache, and the following code snipped is a Spectre v4 exploit as explained in Section 2.2.6.

The only difference is that the pointer sanitization mechanism is not applied since we are able to bypass it.

We conclude that there is no need to write the malicious pointer to the stack slot in the current execution. The behavior could be exploited using a variable offset stack address, which then loads a slow value. The store is then bypassed, and the value from the stack slot is read. Since we control the value, we can use this to access any memory location and leak kernel data via a side-channel. This attack vector was fixed in another context [42]. With variable stack offset, it was possible to leak uninitialized stack data containing sensitive kernel pointers. To prevent the leakage, a patch prohibited variable stack offset. Unfortunately, we are unable to bypass the restrictions. We could not get a slow read to force the speculative store bypass inside the CPU core. Nevertheless, if this limitation gets overcome, or if there is a way to skip a particular store, this is exploitable.

While finishing the thesis, another vector was discovered. It might be enough to push and pop the register to the stack, to force the CPU to speculate far enough to execute a Spectre gadget. First, however, it has to be verified in further research.

6.2.2 Proposed Solution

Since we already have a speculative store bypass counter measurement, the code changes necessary to eliminate this issue are small. The preventive store of zero is already applied if the stack slot was previously used. However, if the stack slot was not used, the countermeasure is not applied. To fix this, we only have to change the stack slot condition to sanitize uninitialized slots. Due to another store bypass vector discovered, which can bypass the mitigation, the patch is not released at the time. It may change in the future.

6.3 CVE-2021-33624

This attack is a speculative type confusion, but the vector can also be used to achieve an out-of-bounds memory read.

6.3.1 Attack

Here the basic idea from Spectre v1 is taken to create a speculative type confusion. We have two distinct branches which get either both executed or none of them. That is only true for the non-speculative domain. If the first jump is taken and the second jump is not taken in the speculative context, we can use this to gain speculative kernel memory access. Notice that other proof-of-concept variants

use two mutually exclusive branches where either the first or the second branch gets executed and never both. Here the verifier is miss trained not to take both branches.

Assume we have the eBPF assembly shown in Listing 6.4.

```
1  (bf) r4 = r2        //set r4 to the leak address (scalar)
2  (15) if r5 == 0x0 goto pc+1
3  (bf) r4 = r0        //assign r4 a map pointer
4  [...]
5  (79) r3 = *(u64 *)(r0 +4608) //slowly load r3
6  (57) r3 &= 1
7  (57) r3 &= 2
8  (0f) r5 += r3       //add r3, known 0, to r5
9  (15) if r5 == 0x0 goto pc+12
10 (71) r4 = *(u8 *)(r4 +0) //speculatively access memory
```

Listing 6.4: CVE-2021-33624 proof-of-concept code

As we can see, if register r5 is equal to 0, both conditional gotos are taken. However, if r5 is not equal to 0, both are not taken. Then we overwrite r4 with a valid pointer and load a value from that map into r4 (line 10). Since r5 has not changed between both branches, the verifier assumes that both jumps are taken or none of them, which is correct for the non-speculative domain. However, if we imagine that the first branch is taken and the second is not taken, we would dereference a scalar value and load the value in r4. We would have arbitrary memory access. However, this is not possible in the non-speculative domain, but in the speculative environment, it could occur. Since the CPU state is reset to the pre-speculative state if the branch prediction was wrong, we use a Spectre-like gadget to obtain the secret information and manipulate the cache to retrieve the data from the speculative context.

This code does not work out of the box since the branch prediction units in modern CPUs are intelligent. The branch predictor would detect that either both jumps are taken or none of them. So if the first jump is taken, the predictor would assume that we will also take the second. It is done by relying on the Global History Record and the Pattern History Table. Evtyushkin et al. have reversed this partially [43]. So we need to find a way that the CPU is not relying on the branch prediction unit or precisely manipulate that unit. We achieved this by putting many conditional jump instructions in the gap between instructions 4 and 5, which are not taken and manipulate the branch predictor. It works reasonably well in most cases, but sometimes the CPU seems to detect the fraud, and there are no more reliable results.

However, Piotr Krysiuk has discovered that using a long sequence of *DIV* instructions before the second branch influences the decisions of the branch predictor more

precisely. The results are empirically verified, and the success rate is nearly 100%. The reason for it is that *DIV* instructions get internally patched by the CPU to prevent division by 0 exceptions. Each *DIV* instruction also contains a branch instruction, which manipulates the branch predictor. Also, it includes the actual *DIV* instruction and code to save the remainder of the division. Thus, a *DIV* instruction gets internally patched into ten eBPF instructions.

Since the internal of Intel CPUs is proprietary, there is no way to confirm this, but the experiments show other branch instructions in front of the branch we want to manipulate influence the prediction for that specific branch.

The issue was also discovered by Ofek Kirzner, Adam Morrison, and Piotr Krysiuk [38]

6.3.2 Fix

This vulnerability gets patched by prohibiting the verifier from making assumptions about if jumps are taken or not taken in the speculative domain. These assumptions were used as an optimization technique, that all paths which are not reachable in the non-speculative domain got pruned, thus are not checked. In addition, the patch introduced an individual check for speculative domain paths for unprivileged users. For example, if we try to use a register as a pointer in one case and the other case the same register as a scalar, as we do in 6.4, the verifier will reject the program since it can detect that in the speculative domain we are treating a scalar value as a pointer and dereferencing it [44].

6.4 Speculative Store Bypass

Piotr Krysiuk discovered this attack vector, and we have extended his proof-of-concept in the first place to work without a patched kernel. This attack bypasses the current speculative store bypass counter measurements entirely.

6.4.1 Attack

While discussing the theoretical, speculative store bypass issue and sending Piotr Krysiuk the findings, he came up with another attack vector unrelated to the attack mentioned above. The idea bypasses the current mitigation against Spectre v4. Instead of relying on a slow memory load, he fills up the processor's store queue so no dependency can be detected when the dependent load is executed. It works because Intel CPUs have a different store and load port. For Skylake, we have two load ports and one store port. A long sequence of store instructions is used to fill the buffer, then the critical path starts, and the store we want to skip comes. After

the CPU skips the store, the dependent load is executed. Since the store buffer is filled up and the CPU does not want to stall, the load is performed out of order speculating there is no dependency between any previous write. The root cause seems to be that the store queue is full, and no forwarding can happen to detect the RAW dependency. The proof-of-concept code is extended to work even without an unpatched kernel. Of course, it is only a hypothesis that the store buffer queue is overcrowded and the dependency is not resolved correctly. However, this hypothesis seems reasonable for now.

6.4.2 Proposed Solutions

This issue is difficult to fix in Software since we cannot sanitize the slow store with another preceding fast store. Both stores will be bypassed and thus not get executed in the speculative domain. Nonetheless, there are three options available that bring major performance or verification restrictions with them.

1. Use the `LFENCE` instruction after any pointer gets stored on the stack. Depending on the use case, this does not impact performance as much. However, there might be other vectors to bypass the insertion of the `LFENCE` instruction. Also, the problem, how does this affects other architectures without the `LFENCE` instruction persists.
2. Disable speculative store bypass entirely on the execution thread. This would prevent any Spectre v4 attacks against unprivileged eBPF at the cost of the highest performance penalty, affecting many programs not using the stack.
3. The last option is to disallow any pointer to be stored on the map. It could break userspace applications, but the compiler can bypass this restriction. The only pointer, which has to be preserved all time, is the argument `CTX`, which gets passed at the beginning of the program. All other pointers can be retrieved dynamically.

We would suggest using the 3. possibility because it has the lowest performance impacts and is a general-purpose solution for all architectures. In addition, the problem with userspace compatibility can be bypassed by adjusting the `clang` compile rules.

7 Conclusion

Similar to modern cryptography, one of the most significant threats is not the mathematical concept used, instead side-channels, which allow obtaining data from the running system. Similarities can be seen in eBPF. While the verifier is currently not perfect, it is a pretty solid and well-written piece of software, which got attacked hard with side-channel attacks in late 2020 and 2021. While these vulnerabilities can not be used for privilege escalation, they are the greatest threat for information leakage within the kernel.

7.1 Summary

The fuzzing attempts were not as successful as expected, but the manual inspections and the resulting speculative attacks against eBPF are a solid base for this thesis. We have shown how the fuzzer performs on eBPF. Furthermore, we evaluated the results and discussed how eBPF-fuzzing could be improved.

We have also shown practical attacks on eBPF and how to implement mitigations to prevent them. A total of three speculative execution issues were found, and another speculative issue was supportive, accompanied by writing the proof-of-concept code for an unpatched kernel. Two CVE's got assigned for the speculative bounds check bypass and the speculative type confusion. Furthermore, two speculative store bypass issues are still under investigation by the maintainers, and no patches have been released until now. Both got a CVE reserved. The thesis also discussed the countermeasures which could be applied against the attacks. Lastly, we introduced the actual patches which prevent the issues from being exploited.

7.2 Future Work

eBPF is unique in the operating system landscape. It consists of a static code analyzer, which checks code that gets loaded into the Linux kernel. To improve the security model of eBPF, we present three more research fields where one can look upon.

7.2.1 Formal Methods

One problem we have encountered getting into the verifier is that the restrictions that apply to the program were not specified mathematically. There are only a few blog posts that cover some properties. Also, the checks in the verifier are not as easy to understand. This section introduces ways how specific properties could be defined for a better analysis. If the program gets translated into a directed acyclic graph (DAG), we can use this representation for `Model Checking` purposes. However, not all properties can be defined that way, and we still have to deal with mathematical restrictions, for example, the state explosion problem.

The property that all execution path should end with a exit instruction can be formalized with a CTL formula $\mathbf{A}!\texttt{exit}\mathbf{U}(\texttt{exit} \wedge \mathbf{AX}\varnothing)$. As we can see, even for such a simple statement, the formulas become complicated. Nonetheless, the computer can more easily read the formula and check the program against it. The problem is the properties that are not expressable in linear temporal logic (LTL) or computation tree logic (CTL). For example *At any time, at least as many coins have been inserted as bottles are given away.* Since an automat who implements this formula must count up to infinity and thus needs infinite states, which is not possible. Luckily eBPF is not **Turing Complete** due to its limitation in program size. It follows that the abovementioned problems could be overcome, and a model checking system could be designed for eBPF. However, the state-explosion problem persists; even with the symbolic model checking approach, we cannot check as many states as an eBPF could generate. Thus a precise mathematical model for eBPF needs to be developed, so it is possible to prove that programs are safe.

7.2.2 New Fuzzing Techniques

Scannell has already shown that using a priori knowledge and searching for specific bugs could lead to a drastic improvement in fuzzing success. While discussing the results of his work with him, he came up with an exciting idea. As we have already seen, neither syzkaller nor kAFL could find bugs in eBPF, and we conclude that one of the reasons for it might be that we have used the wrong sanitizer. His idea is that one could program a unique eBPF sanitizer, which uses the information statically determined by the verifier and then checks it against the actual values within the running program. Having a value outside of the range, which the verifier has calculated, or having bits (un)-set, where the verifier thinks the bits are flipped, directly led to a local privilege escalation issue. Thus, fuzzing for such issues is in terms of security analysis the best one can do.

However, what sounds so simple is complex kernel programming and requires much experience and time. Also, significant code changes within the verifier and JIT compiler or interpreter, depending on what we want to fuzz, are necessary to get this

working. Thus, the chances that such a project gets upstream are small, and maintaining it out of tree is probably not worth it. Nonetheless, it would be interesting how such a sanitizer performs.

7.2.3 Advandced Microarchitectural Exploits

At the time of writing, the only speculative attacks against eBPF are the standard variants, which are easy to understand and relatively easy to exploit. Another class of attacks was uncovered in later research, which could be a significant thread against eBPF. These are specifically the MDS vulnerabilities [45][46]. These attacks are even more challenging to exploit than the classic Spectre vulnerabilities, and a deeper understanding of the underlying processor technology is necessary. Currently, there are no exploits out there, which attack eBPF specifically, so a new attack vector needs to be crafted, bypassing all restrictions that eBPF applies to the programs.

However, eBPF programs run in kernel mode and thus counter measurements against these attacks like kernel page table isolation (KPTI) [47] or special scheduling techniques, to avoid that threads of different privilege levels are scheduled together [48], could not prevent the BPF program from leaking data. It is most likely, that eBPF can be used for such attacks, as long as the microcode updates do not prevent them. However, even newer CPUs are not resistant against speculative attacks in general because it often requires a performance penalty if they get applied. AMD recently published a whitepaper, which states that their new CPUs are vulnerable against Spectre v4 attacks [49]. Similar to Intel, this performance optimization can be disabled, but the main problem persists. There is currently no perfect solution to mitigate all side channels, and unprivileged eBPF could allow us to exploit even more variants in the future.

A Acronyms

JIT just-in-time

CPU central processing unit

ISA instruction set architecture

CVE Common Vulnerabilities and Exposures

WAR write-after-read

RAW read-after-write

WAW write-after-write

CCX core/compute complex

tnum tristate number

msb most significant bit

BTF BPF Type Information

KPTI kernel page table isolation

KASLR kernel address space layout randomization

XDP Xpress Data Protocol

LSM Linux Security Module

CTL computation tree logic

LTL linear temporal logic

DAG directed acyclic graph

B Appendix

B.1 Working with eBPF

Working with eBPF requires a toolchain to obtain the information necessary to analyze the security concept. All code for Listings, which shows BPF pseudocode, was obtained using the command `sudo bpftool prog dump xlated id []`. To get an entire list of loaded BPF programs, run `sudo bpftool prog show`. If we want to obtain the same information for maps we use the command `sudo bpftool map show`. To get more specific information, we recommend reading the man page of bpftool. To get a complete list of available functions, maps and programs, use `bpftool feature probe`.

The verifier also consists of different log levels. While discussing with Daniel Borkmann, we conclude that even the highest level might not be enough for security analysis. The highest level can be set with the debug level variable to 7 [50] as seen in Listing B.1.

```
union bpf_attr create_prog_attrs = {
    [...]
    .log_level = 7,
    [...]
};
```

Listing B.1: BPF loglevel

The BPF tree is under rapid development; all information in the thesis is provided as of July 22, 2021. Many things will likely change in the future. The crucial information is cited with resources from the corresponding Linux kernel commit.

The macros for C programming can be obtained from the header file `linux/include/Linux/filter.h`. All eBPF assembly macros follow the Intel syntax, `opcode | dst | src`, where memory instructions could also contain an offset value. This could be changed if one decides to write his own code but the macros within the kernel use this semantic.

The proof-of-concept codes for the exploits and further code snippets used can be found under `https://github.com/Kakashiiiiy/bachelors-thesis`.

B.2 Spectre Masking Example

```
Index    000000000000003B
MaxValue 000000000000003C
-------------------------
Minus    0000000000000001
Or       000000000000003B
Neg      FFFFFFFFFFFFFFC5
Shift    FFFFFFFFFFFFFFFF
Result   000000000000003B
```

Listing B.2: Pointer offset masking in bounds

```
Index    000000000000003D
MaxValue 000000000000003C
-------------------------
Minus    FFFFFFFFFFFFFFFF
Or       FFFFFFFFFFFFFFFF
Neg      0000000000000001
Shift    0000000000000000
Result   0000000000000000
```

Listing B.3: Pointer offset masking out of bounds

B.3 Spectre Proof of Concept Result

```
$ sudo cat /proc/kallsyms | grep core_pattern
 ffffffff85d23980  D core_pattern
$ cat /proc/sys/kernel/core_pattern
|/usr/lib/systemd/systemd-coredump %P %u %g %s %t %c %h
```

Listing B.4: Additional leak information

Table B.1: Spectre leakage result

ffffffff85d23990	64	2f	73	79	73	74	65	6d	64	2d	63	6f	72	65	64	75
ffffffff85d23980	7c	2f	75	73	72	2f	6c	69	62	2f	73	79	73	74	65	6d
ffffffff85d239a0	6d	70	20	25	50	20	25	75	20	25	67	20	25	73	20	25
ffffffff85d239b0	74	20	25	63	20	25	68	00	00	00	00	00	00	00	00	00
ffffffff85d239c0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
ffffffff85d239d0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
ffffffff85d239e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
ffffffff85d239f0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
ffffffff85d23a00	a0	36	d2	85	ff	ff	ff	ff	a0	3a	d2	85	ff	ff	ff	ff
ffffffff85d23a10	20	08	db	85	ff	ff	ff	ff	0e	9e	59	85	ff	ff	ff	ff
ffffffff85d23a20	40	7a	d1	85	ff	ff	ff	ff	a0	25	d5	85	ff	ff	ff	ff
ffffffff85d23a30	d0	3a	d2	85	ff	ff	ff	ff	d0	36	d2	85	ff	ff	ff	ff
ffffffff85d23a40	d1	03	00	00	00	00	00	00	a0	81	ce	85	ff	ff	ff	ff
ffffffff85d23a50	13	db	57	85	ff	ff	ff	ff	00	00	00	00	00	00	00	00
ffffffff85d23a60	00	00	00	00	00	00	00	00	40	3b	d2	85	ff	ff	ff	ff

Table B.2: Spectre leakage result ascii

```
||/usr/lib/system|
|d/systemd-coredu|
|mp %P %u %g %s %|
|t %c %h.........|
|................|
|................|
|................|
|................|
|.6.......:......|
| .........Y.....|
|@z.......%......|
|.:.......6......|
|................|
|..W.............|
|........@;......|
```

B.4 BPF Structures

Table B.3: eBPF program types

BPF_PROG_TYPE_UNSPEC
BPF_PROG_TYPE_SOCKET_FILTER
BPF_PROG_TYPE_KPROBE
BPF_PROG_TYPE_SCHED_CLS
BPF_PROG_TYPE_SCHED_ACT
BPF_PROG_TYPE_TRACEPOINT
BPF_PROG_TYPE_XDP
BPF_PROG_TYPE_PERF_EVENT
BPF_PROG_TYPE_CGROUP_SKB
BPF_PROG_TYPE_CGROUP_SOCK
BPF_PROG_TYPE_LWT_IN
BPF_PROG_TYPE_LWT_OUT
BPF_PROG_TYPE_LWT_XMIT
BPF_PROG_TYPE_SOCK_OPS
BPF_PROG_TYPE_SK_SKB
BPF_PROG_TYPE_CGROUP_DEVICE
BPF_PROG_TYPE_SK_MSG
BPF_PROG_TYPE_RAW_TRACEPOINT
BPF_PROG_TYPE_CGROUP_SOCK_ADDR
BPF_PROG_TYPE_LWT_SEG6LOCAL
BPF_PROG_TYPE_LIRC_MODE2
BPF_PROG_TYPE_SK_REUSEPORT
BPF_PROG_TYPE_FLOW_DISSECTOR
BPF_PROG_TYPE_CGROUP_SYSCTL
BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE
BPF_PROG_TYPE_CGROUP_SOCKOPT
BPF_PROG_TYPE_TRACING
BPF_PROG_TYPE_STRUCT_OPS
BPF_PROG_TYPE_EXT
BPF_PROG_TYPE_LSM
BPF_PROG_TYPE_SK_LOOKUP

Table B.4: eBPF map types

BPF_MAP_TYPE_UNSPEC
BPF_MAP_TYPE_HASH
BPF_MAP_TYPE_ARRAY
BPF_MAP_TYPE_PROG_ARRAY
BPF_MAP_TYPE_PERF_EVENT_ARRAY
BPF_MAP_TYPE_PERCPU_HASH
BPF_MAP_TYPE_PERCPU_ARRAY
BPF_MAP_TYPE_STACK_TRACE
BPF_MAP_TYPE_CGROUP_ARRAY
BPF_MAP_TYPE_LRU_HASH
BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_MAP_TYPE_LPM_TRIE
BPF_MAP_TYPE_ARRAY_OF_MAPS
BPF_MAP_TYPE_HASH_OF_MAPS
BPF_MAP_TYPE_DEVMAP
BPF_MAP_TYPE_SOCKMAP
BPF_MAP_TYPE_CPUMAP
BPF_MAP_TYPE_XSKMAP
BPF_MAP_TYPE_SOCKHASH
BPF_MAP_TYPE_CGROUP_STORAGE
BPF_MAP_TYPE_REUSEPORT_SOCKARRAY
BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE
BPF_MAP_TYPE_QUEUE
BPF_MAP_TYPE_STACK
BPF_MAP_TYPE_SK_STORAGE
BPF_MAP_TYPE_DEVMAP_HASH
BPF_MAP_TYPE_STRUCT_OPS
BPF_MAP_TYPE_RINGBUF
BPF_MAP_TYPE_INODE_STORAGE
BPF_MAP_TYPE_TASK_STORAGE

Table B.5: eBPF register types

NOT_INIT
SCALAR_VALUE
PTR_TO_CTX
CONST_PTR_TO_MAP
PTR_TO_MAP_VALUE
PTR_TO_MAP_VALUE_OR_NULL
PTR_TO_STACK
PTR_TO_PACKET
PTR_TO_PACKET_META
PTR_TO_PACKET_END
PTR_TO_FLOW_KEYS
PTR_TO_SOCKET
PTR_TO_SOCKET_OR_NULL
PTR_TO_SOCK_COMMON
PTR_TO_SOCK_COMMON_OR_NULL
PTR_TO_TCP_SOCK
PTR_TO_TCP_SOCK_OR_NULL
PTR_TO_TP_BUFFER
PTR_TO_XDP_SOCK
PTR_TO_BTF_ID
PTR_TO_BTF_ID_OR_NULL
PTR_TO_PERCPU_BTF_ID
PTR_TO_MEM
PTR_TO_MEM_OR_NULL
PTR_TO_RDONLY_BUF
PTR_TO_RDONLY_BUF_OR_NULL
PTR_TO_RDWR_BUF
PTR_TO_RDWR_BUF_OR_NULL
PTR_TO_FUNC
PTR_TO_MAP_KEY

Table B.6: eBPF system call arguments

BPF_MAP_CREATE
BPF_MAP_LOOKUP_ELEM
BPF_MAP_UPDATE_ELEM
BPF_MAP_DELETE_ELEM
BPF_MAP_GET_NEXT_KEY
BPF_PROG_LOAD
BPF_OBJ_PIN
BPF_OBJ_GET
BPF_PROG_ATTACH
BPF_PROG_DETACH
BPF_PROG_TEST_RUN
BPF_PROG_GET_NEXT_ID
BPF_MAP_GET_NEXT_ID
BPF_PROG_GET_FD_BY_ID
BPF_MAP_GET_FD_BY_ID
BPF_OBJ_GET_INFO_BY_FD
BPF_PROG_QUERY
BPF_RAW_TRACEPOINT_OPEN
BPF_BTF_LOAD
BPF_BTF_GET_FD_BY_ID
BPF_TASK_FD_QUERY
BPF_MAP_LOOKUP_AND_DELETE_ELEM
BPF_MAP_FREEZE
BPF_BTF_GET_NEXT_ID
BPF_MAP_LOOKUP_BATCH
BPF_MAP_LOOKUP_AND_DELETE_BATCH
BPF_MAP_UPDATE_BATCH
BPF_MAP_DELETE_BATCH
BPF_LINK_CREATE
BPF_LINK_UPDATE
BPF_LINK_GET_FD_BY_ID
BPF_LINK_GET_NEXT_ID
BPF_ENABLE_STATS
BPF_ITER_CREATE
BPF_LINK_DETACH
BPF_PROG_BIND_MAP

List of Figures

2.1 Intel Core i5-8250u topology 6
2.2 Instruction encoding eBPF assembly 10
2.3 Calling convention eBPF assembly 11
2.4 Loading process of an eBPF program 13
2.5 `BPF_MAP_TYPE_ARRAY` layout 15

4.1 High-level execution flow of the eBPF Spectre v1 exploit code 30
4.2 High-level execution flow of the eBPF Spectre v4 exploit code 31

5.1 Path coverage `BPF_PROG_LOAD` 34
5.2 Path coverage `BPF_MAP_CREATE` 35

List of Tables

2.1 MESI concurent states table 7

B.1 Spectre leakage result 55
B.2 Spectre leakage result ascii 55
B.3 eBPF program types 56
B.4 eBPF map types 57
B.5 eBPF register types 58
B.6 eBPF system call arguments 59

List of Listings

2.1 Spectre v1 example 4
2.2 Spectre v4 example 5
2.3 BPF system call wrapper 9
2.4 Verification restrictions from `linux/kernel/bpf/verifier.c` 12
2.5 `BPF_MAP_CREATE` struct 14
2.6 eBPF Spectre v1 example 18
2.7 eBPF pointer sanitization 18
2.8 eBPF Spectre v4 example 19
2.9 eBPF Spectre v4 countermeasure 19
2.10 `ioctl` eBPF attachment system calls 20

5.1 `BPF_PROG_LOAD` agent 33
5.2 `BPF_MAP_CREATE` agent 35

6.1 CVE-2021-29155 proof-of-concept code 39
6.2 Rewrite probability stack offset `-424` 42
6.3 CVE-2021-34556 assisted proof-of-concept code 42
6.4 CVE-2021-33624 proof-of-concept code 44

B.1 BPF loglevel 53
B.2 Pointer offset masking in bounds 54
B.3 Pointer offset masking out of bounds 54
B.4 Additional leak information 54

Bibliography

[1] Alexei Starovoitov, "Merge branch 'bpf-unprivileged'." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c1bf5fe03184f782f2a6827cf314ae58834865da, as of July 22, 2021.

[2] Piotr Krysiuk, "Cve-2020-27170." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27170, as of July 22, 2021.

[3] Piotr Krysiuk, "Cve-2020-27171." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27171, as of July 22, 2021.

[4] Jann Horn, "Issue 1711: Linux: ebpf spectre v1 mitigation is insufficient." https://bugs.chromium.org/p/project-zero/issues/detail?id=1711, as of July 22, 2021.

[5] Jann Horn, "Issue 1528: speculative execution, variant 4: speculative store bypass." https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, as of July 22, 2021.

[6] Manfred Paul, "Cve-2020-8835: Linux kernel privilege escalation via improper ebpf program verification." https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification, as of July 22, 2021.

[7] Lucas Leong, Manfred Paul, "Cve-2021-31440: An incorrect bounds calculation in the linux kernel ebpf verifier." https://www.zerodayinitiative.com/blog/2021/5/26/cve-2021-31440-an-incorrect-bounds-calculation-in-the-linux-kernel-ebpf-verifier, as of July 22, 2021.

[8] Simon Scannell, "Fuzzing for ebpf jit bugs in the linux kernel." https://scannell.io/posts/ebpf-fuzzing/, as of July 22, 2021.

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

[10] Matt Miller, "Analysis and mitigation of speculative store bypass (cve-2018-3639)." https://msrc-blog.microsoft.com/2018/05/21/analysis-and-

mitigation-of-speculative-store-bypass-cve-2018-3639/, as of July 22, 2021.

[11] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[12] AMD, "Tuning guide for amd epyc™ 7002 series processors." https://developer.amd.com/wp-content/resources/56827-1-0.pdf, as of July 22, 2021.

[13] N. R. Mahapatra and B. Venkatrao, "The processor-memory bottleneck: problems and solutions," *Crossroads*, vol. 5, no. 3es, p. 2, 1999.

[14] M. E. Thomadakis, "The architecture of the nehalem processor and nehalem-ep smp platforms," *Resource*, vol. 3, no. 2, pp. 30–32, 2011.

[15] AMD, "Amd64 architecture programmer's manual, volume 2: System programming." https://web.archive.org/web/20170619232736/http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf, as of July 22, 2021.

[16] AMD, "Software optimization guide for amd family 19h processors (pub)." https://www.amd.com/system/files/TechDocs/56665.zip, as of July 22, 2021.

[17] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th annual international symposium on Computer architecture*, pp. 348–354, 1984.

[18] iovisor project, "Unofficial ebpf spec." https://github.com/iovisor/bpf-docs/blob/master/eBPF.md, as of July 22, 2021.

[19] Wang YanQing, "bpf, x86-32: add ebpf jit compiler for ia32." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=03f5781be2c7b7e728d724ac70ba10799cc710d7, as of July 22, 2021.

[20] Alexei Starovoitov, "bpf, capabilities: introduce cap_bpf." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a17b53c4a4b55ec322c132b6670743612229ee9c, as of July 22, 2021.

[21] Thomas Gleixner, "bpf: Make bpf and preempt_rt co-exist." https://lwn.net/Articles/812503/, as of July 22, 2021.

[22] Alexei Starovoitov, "bpf: introduce bounded loops." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2589726d12a1b12eaaa93c7f1ea64287e383c7a5, as of July 22, 2021.

[23] Alexei Starovoitov, "bpf: increase complexity limit and maximum program size." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82, as of July 22, 2021.

[24] Daniel Borkmann, "bpf: prevent out of bounds speculation on pointer arithmetic." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=979d63d50c0c0f7bc537bf821e056cc9fe5abd38, as of July 22, 2021.

[25] Thomas Gleixner, "x86-pti-for-linus." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=35277995e17919ab838beae765f440674e8576eb, as of July 22, 2021.

[26] Alexei Starovoitov, "bpf: Prevent memory disambiguation attack." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=af86ca4e3088fe5eacf2f7e58c01fa68ca067672, as of July 22, 2021.

[27] Alexei Starovoitov, "bpf: introduce bpf-tail-call() helper." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3f55b7ed5e5d4aa7291e3a1e2f7224eeba5810ba, as of July 22, 2021.

[28] Daniel Borkmann, "bpf, x64: implement retpoline for tail call." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a493a87f38cfa48caaa95c9347be2d914c6fdf29, as of July 22, 2021.

[29] Daniel Borkmann, "optimize-bpf_tail_call." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6dbae03b2e851fe58e9d9d1ac9ed58017b07960c, as of July 22, 2021.

[30] Greg Marsden, "Bpf: A tour of program types." https://blogs.oracle.com/linux/notes-on-bpf-1, as of July 22, 2021.

[31] "Syzkaller'." https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, as of July 22, 2021.

[32] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[33] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *NDSS*, vol. 19, pp. 1–15, 2019.

[34] T. Blazytko, M. Bishop, C. Aschermann, J. Cappos, M. Schlögel, N. Korshun, A. Abbasi, M. Schweighauser, S. Schinzel, S. Schumilo, *et al.*, "{GRIMOIRE}: Synthesizing structure while fuzzing," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1985–2002, 2019.

[35] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[36] Linux, "The kernel address sanitizer (kasan)." `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html`, as of July 22, 2021.

[37] Linux, "The undefined behavior sanitizer - ubsan." `https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html`, as of July 22, 2021.

[38] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[39] Dmitry Vyukov, "bpf: undefined shift in __bpf_prog_run." `https://groups.google.com/g/syzkaller/c/H7o2oz9CcKg/m/uzaiF7eqBwAJ`, as of July 22, 2021.

[40] Anatoly Trosinenko, "kbdysch." `https://github.com/atrosinenko/kbdysch`, as of July 22, 2021.

[41] Daniel Borkmann, "bpf: Tighten speculative pointer arithmetic mask." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/patch/kernel/bpf/verifier.c?id=7fedb63a8307dda0ec3b8969a3b233a1dd7ea8e0`, as of July 22, 2021.

[42] Daniel Borkmann, "bpf: Fix leakage of uninitialized bpf stack under speculation." `https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/patch/?id=801c6058d14a82179a7ee17a4b532cac6fad067f`, as of July 22, 2021.

[43] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[44] Daniel Borkmann, "bpf: Fix leakage under speculation on mispredicted branches." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9183671af6dbf60a1219371d4ed73e23f43b49db`, as of July 22, 2021.

[45] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2019.

[46] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[47] Thomas Gleixner, "Merge branch 'x86-pti-for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/tip/tip." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5aa90a84589282b87666f92b6c3c917c8080a9bf`, as of July 22, 2021.

[48] Joel Fernandes, "Documentation: Add usecases, design and interface for core scheduling." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0159bb020ca9a43b17aa9149f1199643c1d49426`, as of July 22, 2021.

[49] AMD, "Security analysis of amd predictive store forwarding." `https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf`, as of July 22, 2021.

[50] Alexei Starovoitov, "bpf: add verifier stats and log_level bit 2." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=06ee7115b0d1742de745ad143fb5e06d77d27fba`, as of July 22, 2021.